Test en ligne pour la détection des fautes intermittentes dans les architectures multiprocesseurs embarquées

Julien Guilhemsang

UNIVERSITE DE NICE-SOPHIA ANTIPOLIS - UFR Sciences
École Doctorale STIC

T H E S E

pour obtenir le titre de
Docteur en Sciences
de l'UNIVERSITE de Nice-Sophia Antipolis

Discipline : Électronique

Présentée et soutenue par
Julien GUILHEMSANG

Test en ligne pour la détection des fautes intermittentes dans les architectures multiprocesseurs embarquées

Thèse dirigée par Alain GIULIERI
Soutenue le 8 avril 2011

Jury :

MM. :	Lirida	NAVINER	Professeur à Télécom ParisTech	Présidente
	Bruno	ROUZEYRE	Professeur à l'université de Montpellier 2	Rapporteur
	Thomas	ZIMMER	Professeur à l'université de Bordeaux 1	Rapporteur
	Michel	AUGUIN	Directeur de recherche au CNRS	Examinateur
	Alain	GIULIERI	Professeur à l'université de Nice-Sophia Antipolis	Directeur de thèse
	Olivier	HERON	Ingénieur Chercheur au CEA LIST	Encadrant
	Nicolas	VENTROUX	Ingénieur Chercheur au CEA LIST	Encadrant

À Joël, mon grand père

C'est ce que nous pensons déjà connaître qui nous empêche souvent d'apprendre.

Claude Bernard

Remerciements

Ayant réalisé ma thèse dans le Laboratoire de Fiabilisation des Systèmes Embarqués du CEA LIST, j'y ai tissé de nombreuses amitiés. Ainsi, mes premiers remerciements vont à l'ensemble des personnes que j'ai côtoyé durant mes années au CEA. Je pense en particulier à Fabrice Auzanneau qui m'a accueilli au sein de son laboratoire, aux collègues qui m'ont suivi depuis mon arrivée au CEA, ainsi que ceux que je n'ai croisé que peu de temps. J'espère leur avoir apporté au moins autant qu'ils m'ont donné.

Je remercie mes collègues de bureau, Jean-Marc et Maud, pour la bonne ambiance et le calme qui m'ont permis de mener à bien ma dernière année de thèse.

Je remercie Olivier Gonçalves pour sa contribution dans la mise en œuvre de la plateforme expérimentale au cours de son stage de fin d'étude. Je lui souhaite de mener à bien la thèse qu'il a entrepris.

Je remercie mon directeur de thèse, Alain Giulieri, professeur à l'université de Nice-Sophia Antipolis, pour avoir su m'écouter et me soutenir dans les périodes difficiles.

Je remercie mes encadrants, Nicolas et Olivier, pour leurs multiples relectures, qui ont contribué à la qualité de ce mémoire.

Je remercie Lirida Naviner, professeur à Télécom ParisTech, qui m'a fait l'honneur de présider mon jury de thèse.

Je remercie Bruno Rouzeyre, professeur à l'université de Montpellier 2, et Thomas Zimmer, professeur à l'université de Bordeaux 1, d'avoir accepté la charge de rapporteur.

Je remercie Michel Auguin, directeur de recherche CNRS à l'université de Nice-Sophia Antipolis, d'avoir accepté de faire partie de mon jury de thèse. Depuis mon Master, il compte parmi ceux qui m'ont donné le goût de la recherche.

Je remercie mes parents pour m'avoir donné le goût des sciences et finalement l'envie de réaliser cette thèse. Ils ont toujours été là, avec mes sœurs, quand j'en avais besoin et ils ont su m'apporter un soutien inconditionnel. De même je remercie, tous les membres de ma famille, qui bien que non scientifiques, ont eut le courage de relire ma thèse.

Pour finir, je remercie tout particulièrement ma femme Alice pour m'avoir toujours soutenu, et pour avoir supporté mes années de thèse et mes longues nuits de rédaction ...

Table des matières

Table des figures

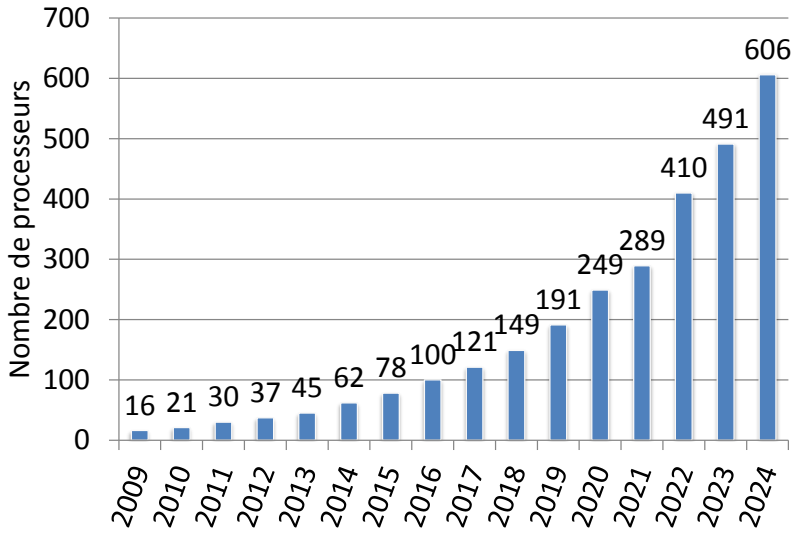
Liste des tableaux

Introduction

Aujourd'hui, les systèmes embarqués sont partout et requièrent de plus en plus de puissance de calcul. Pour cela, ces dernières années, les progrès d'intégration ont permis de diminuer la taille des transistors, d'augmenter la fréquence de fonctionnement et de diminuer la tension d'alimentation. Tout cela en augmentant progressivement le nombre de processeurs dans une même puce. En effet, seules les architectures multiprocesseur semblent pouvoir suivre l'évolution des applications, en apportant le bon compromis entre performances et consommation. Cela est confirmé par les estimations ITRS [1] qui prévoient une augmentation progressive du nombre de processeurs dans une même puce. En particulier, il est prévu des architectures contenant plus de 200 processeurs d'ici seulement dix ans comme le montre la figure suivante. Cela justifie notre intérêt pour ce type d'architecture.

Cependant cette évolution a un impact négatif sur la fiabilité. Les systèmes deviennent de plus en plus sensibles à leur environnement, ce qui conduit à un taux plus important des fautes transitoires. D'autre part, les variations de procédés de fabrication dans les technologies actuelles induisent de plus en plus de variations à l'intérieur même des composants. Ce phénomène, combiné à la hausse des tensions d'alimentation et de la température, augmente progressivement la probabilité d'occurrence des fautes intermittentes voire permanentes. De plus, le vieillissement des composants semble s'accélérer avec la diminution du facteur d'échelle et la diminution non uniforme des tensions d'alimentation, causant l'apparition prématurée des fautes intermittentes voire permanentes.

Si concernant les fautes transitoires et permanentes nous bénéficions d'études expérimentales détaillées, ce n'est pas le cas des fautes intermittentes. Or, pour tenter de se prémunir de ces fautes, il est important de comprendre leur comportement, ainsi que leur impact sur le système et les applications.



Nombre de processeurs sur une même puce prévus par l'ITRS [1]

Objectifs

Le premier objectif de ce mémoire est de proposer une plateforme expérimentale capable d'observer des erreurs intermittentes. Il n'existe à notre connaissance aucune étude décrivant précisément ce type d'erreur dans des systèmes actuels. Ainsi, l'étude expérimentale est inévitable si nous voulons comprendre comment détecter, voire anticiper l'apparition des fautes intermittentes.

Plus précisément, nous allons pouvoir déterminer quels sont les facteurs prépondérants dans l'apparition des fautes intermittentes. La question se pose en particulier pour l'activité des processeurs : est-ce que deux processeurs soumis à des activités différentes ont un taux d'erreurs intermittentes équivalent ? De plus, si ces fautes ne sont pas destructives, le système peut-il être régénéré ?

Les réponses à ces questions vont permettre de déterminer et d'adapter au mieux un système de détection en ligne des erreurs intermittentes à une architecture multiprocesseur. En effet, il existe une littérature importante concernant la détection des erreurs dans les architectures sur puce. Malheureusement, l'étude de ces méthodes montre qu'aucune technique n'est adaptée à la fois aux systèmes multiprocesseurs et à la détection des erreurs intermittentes. Ainsi, le deuxième objectif de ce mémoire est de développer une méthode de test en ligne répondant à ces deux critères.

Concevoir une telle méthode se heurte à deux contraintes principales : il faut s'assurer que l'efficacité de la détection est suffisante et que l'impact sur les contraintes temps-réel des applications est minime. La validation de notre méthode de détection sera décomposée en deux parties. La première partie sera basée sur l'étude de la probabilité de détection à partir d'un modèle statis-

tique. La deuxième partie sera basée sur l'étude de l'impact du mécanisme de détection sur les applications à partir de simulations sur une plateforme matérielle.

Nous pourrons finalement conclure que la méthode présentée est une technique efficace de détection en ligne des erreurs intermittentes et qu'elle est adaptée aux systèmes multiprocesseurs embarqués.

Organisation du manuscrit

Pour arriver à ces conclusions, le mémoire sera composé de cinq chapitres. Les deux premiers chapitres aborderont les problèmes de fiabilité dans les systèmes actuels, et plus particulièrement ceux relatifs aux fautes intermittentes. Les trois derniers chapitres présenteront une méthode de détection en ligne des ces erreurs adaptée aux architectures multiprocesseur. La figure présente à la fin de cette introduction reprend cette organisation.

Le premier chapitre décrira les notions de base de fiabilité qui seront nécessaires pour comprendre les enjeux des techniques de tolérance aux fautes. Il détaillera les problèmes de fiabilité des composants actuels, en partant des problèmes liés à leur fabrication, jusqu'aux problèmes liés au vieillissement de leurs structures internes. L'étude de ces phénomènes formera les bases théoriques de l'approche expérimentale présentée dans le deuxième chapitre. En particulier, cela permettra de relier la température des composants avec les problèmes de vieillissement et l'apparition de fautes intermittentes.

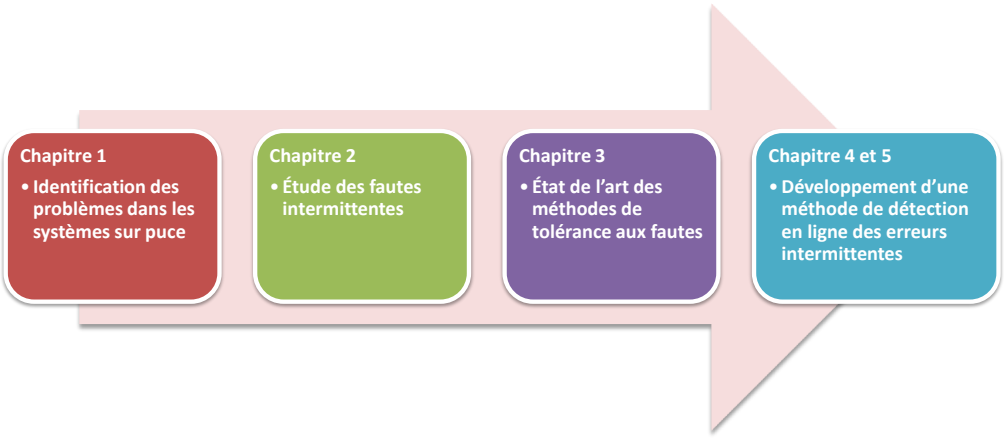
Le deuxième chapitre détaillera la plateforme expérimentale mise en œuvre pour étudier les fautes intermittentes. Cette plateforme se base sur le vieillissement de plusieurs cartes à base de processeurs en 65 nm. Les résultats présentés dans ce chapitre permettront de définir les contraintes liées à la détection des erreurs intermittentes dans les architectures embarquées. En particulier, nous décrirons les modes d'apparition et de disparition spécifiques à ces erreurs. Les observations effectuées dans ce chapitre serviront de base théorique aux chapitres suivants.

Le troisième chapitre présentera les notions de tolérance aux fautes ainsi qu'un état de l'art des techniques de détection en ligne des erreurs. L'objectif de ce chapitre est de sélectionner une méthode capable de détecter des erreurs intermittentes et pouvant être adaptée aux architectures multiprocesseurs. En effet, il n'existe à notre connaissance dans la littérature aucune technique adaptée en même temps aux deux critères précédents. Ainsi, l'efficacité des méthodes sélectionnées par ce chapitre devront être évaluées pour la détection des erreurs intermittentes. Ce sera le rôle des deux derniers chapitres.

Le quatrième chapitre présentera une modélisation statistique des erreurs intermittentes basée sur les observations réalisées dans le deuxième chapitre. Cette modélisation permettra de définir une métrique permettant de comparer différentes implémentations de tests périodiques. En effet, un test périodique peut détecter des erreurs intermittentes mais son efficacité dépend du respect de la période de test. Or, contrairement aux approches de la littérature, nous envisageons d'utiliser un test pseudo-périodique pour réduire l'impact du test dans les architectures multipro-

cesseurs. Dans ce sens, la métrique basée sur la probabilité de détection du test permettra de valider théoriquement l'efficacité des méthodes de tests pseudo-périodiques.

Afin de poursuivre l'étude, le cinquième chapitre détaillera plusieurs implémentations de tests périodiques et pseudo-périodiques sur un cas d'étude d'architecture multiprocesseur matérielle. En particulier, ce chapitre montrera comment l'ordonnancement des tests peut être réalisé en tenant compte de l'occupation des processeurs ou de la priorité des tâches. La simulation de l'architecture, dans différentes conditions de tests et d'applications, permettra de comparer les différents ordonnancements des tests. Ainsi, ce chapitre mettra en évidence une méthode de test permettant de détecter en ligne les erreurs intermittentes dans une architecture multiprocesseur embarquée.



Déroulement du mémoire.

CHAPITRE 1

Introduction à la fiabilité des systèmes sur puce

Sommaire

AUJOURD'HUI les systèmes embarqués sont partout et requièrent de plus en plus de puissance de calcul. Pour cela, ces dernières années, les progrès d'intégration ont permis de diminuer la taille des transistors, d'augmenter la fréquence de fonctionnement et de diminuer la tension d'alimentation. Mais cette évolution a un impact négatif sur la fiabilité.

Les systèmes deviennent de plus en plus sensibles à leur environnement, ce qui conduit à un taux plus important d'erreurs transitoires. D'autre part, les variations de procédés de fabrication dans les technologies actuelles, la hausse des tensions d'alimentation et la température, induisent de plus en plus de variations à l'intérieur même des composants, créant ainsi des sources d'erreurs intermittentes voire permanentes. De plus, le vieillissement des composants semble s'accélérer avec la diminution du facteur d'échelle, causant l'apparition prématurée de ces erreurs. Afin de

s'assurer du bon fonctionnement d'un système il est nécessaire de pouvoir détecter l'apparition d'éventuelles erreurs.

Avant de s'intéresser à la détection des erreurs, il est nécessaire d'introduire quelques bases théoriques de fiabilité et de décrire les différentes sources d'erreurs des circuits intégrés actuels. La première partie de ce chapitre permettra d'une part de définir la fiabilité, et d'autre part d'aborder les notions nécessaires pour estimer la fiabilité d'un système. La deuxième partie décrira les différentes sources d'erreurs auxquelles sont soumis les circuits intégrés. Cette partie débutera par les problèmes de fabrication pour finir par les problèmes de vieillissement. Cela permettra en particulier de relier la température des composants avec les problèmes de vieillissement et l'apparition d'erreurs intermittentes.

1.1 Notions de fiabilité

Le domaine de la fiabilité est vaste et nécessite quelques définitions. Certains termes utilisés dans le langage courant ont des définitions plus précises dans le cadre de la fiabilité, notamment pour les termes *fautes* et *erreurs*.

Dans ce sens, cette partie permettra de détailler quelques définitions de base de la fiabilité. Les différentes phase de la vie d'un circuit intégré seront décrites, et pour finir, cette partie expliquera comment le concepteur d'un système peut en améliorer la disponibilité.

1.1.1 Les différents types de fautes et leur classification

Dans le domaine de la fiabilité les termes *faute/défaut*, erreur et défaillance ont des sens différents et ne doivent pas être confondus.

Une *faute* est induite par un *défaut* physique ou l'accumulation de *défauts* physiques. Une *faute* en s'activant peut engendrer des *erreurs* logiques qui, si elles se propagent, peuvent provoquer une *défaillance* ou une panne du système [2, 3] (voir figure 1.1).

Si l'on prend le cas d'une latch, par exemple, une faute va créer une hausse de tension qui va causer le basculement du niveau de la latch. Du point de vue du circuit, on observe une erreur : le bit est passé de l'état "0" à l'état "1". Si ce bit de donnée est lu par le système, cette erreur peut causer une défaillance. Sinon, on dit que l'erreur est masquée, elle ne cause pas de défaillance.

$$\cdots \longrightarrow \text{Faute} \xrightarrow{activation} \text{Erreur} \xrightarrow{propagation} \text{Défaillance} \xrightarrow{conséquence} \text{Faute} \longrightarrow \cdots$$

FIGURE 1.1 : *Chaîne de défaillance [2]*

Classification des fautes

Les fautes peuvent être classées selon plusieurs critères. Si l'on considère leur durée d'activation, on peut alors différencier les fautes matérielles *permanentes*, *transitoires*, et *intermittentes* (voir figure 1.2). Chacune de ces fautes a des caractéristiques spécifiques qui doivent êtres connues afin de pouvoir les détecter. En effet, pour qu'un système puisse être tolérant, il doit pouvoir détecter chacune d'entre elles. C'est pourquoi cette classification est importante et sera utilisée tout au long de ce mémoire.

Faute permanente : Une *faute permanente* va persister indéfiniment après son occurrence. Ce type de faute peut être induit lors de la fabrication du circuit, et constitue alors les fautes de jeunesse (voir § 1.1.2), ou peut être causé par le vieillissement du circuit.

Faute transitoire : Une *faute transitoire*[4, 5] est une faute qui apparait aléatoirement n'importe où dans le circuit. Sa durée d'apparition est très courte (de l'ordre de la pico-seconde) et sa zone d'apparition très localisée (par exemple quelques cellules d'une mémoire) en fonction de la source de la faute. Ces fautes sont dues à des phénomènes extérieurs qui peuvent être des radiations, des neutrons, des particules alpha, des interférences électromagnétiques, mais aussi des parasites sur l'alimentation. Ce type de faute ne créer pas de dommage dans le circuit, mais le plus souvent des basculements de tension qui en se propageant peuvent inverser des niveaux logiques et modifier la valeur de registres ou de mémoires, et ainsi créer une erreur.

Faute intermittente : Une *faute intermittente* [6–8] peut être latente ou active, selon qu'elle produit une erreur ou non. A la différence des *fautes transitoires*, les *fautes intermittentes* sont reproductibles dans les mêmes conditions de fonctionnement, pour les mêmes données traitées. Contrairement aux deux types de fautes précédentes, il existe très peu d'études publiées sur les fautes intermittentes dans les circuits intégrés actuels. Ces fautes sont essentiellement provoquées par les variations de process (voir § 1.2.2) et par la dégradation du circuit (voir § 1.2.3), combinées à des fluctuations de la température et de la tension d'alimentation [6, 9, 10].

Les erreurs provoquées par des fautes intermittentes peuvent s'activer en *burst* et être maintenues pendant une durée allant de la nano-seconde à la dizaine de secondes [6] suivant la source de l'erreur. Par exemple des fluctuations de la tension d'alimentation peuvent durer de quelques dizaines à quelques centaines de nano-secondes [9, 11, 12], alors que des fluctuations de la température peuvent provoquer des fautes de délais de quelques millisecondes à quelques secondes [13]. Ainsi, le problème des *bursts* de fautes intermittentes n'est pas à négliger.

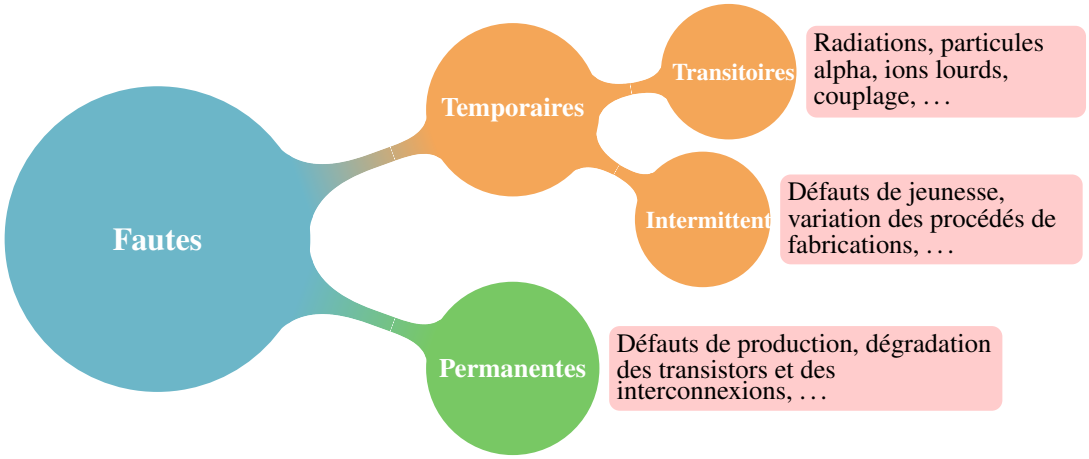


FIGURE 1.2 : *Types de fautes*

1.1.2 La fiabilité au cours de la vie d'un circuit intégré

Définition de la fiabilité

La fiabilité d'un système peut être définie comme son aptitude à assurer une certaine fonctionnalité dans des conditions données et pour un temps donné [3]. En ce qui concerne les System-on-Chip (SoC), les conditions opérationnelles de fonctionnement se résument principalement à la tension d'alimentation et la température de fonctionnement, mais le taux d'humidité, l'altitude ou tout autre paramètre spécifique à un domaine donné peuvent s'ajouter. La durée d'utilisation du système est généralement donnée en fonction de sa durée de vie, qui est estimée de façon statistique sur un ensemble de composants après production.

Comme on le verra par la suite, il est très difficile d'éviter l'apparition d'une faute dans un système électronique actuel. Et, plus la durée de vie d'un système sera grande, plus il sera dit fiable. Ainsi, le niveau de fiabilité d'un composant peut être mesuré à l'aide de son taux de défaillance (*Failure Rate* en anglais). Il est noté λ et est généralement donné en FIT [1]. On définit alors la probabilité que le système soit défaillant après un temps t par une loi exponentielle ayant comme paramètre λ [3] :

$$F(t) = 1 - e^{-\lambda t}$$

Évolution du taux de défaillance

L'évolution du taux de défaillance d'un composant est généralement représenté par une courbe en baignoire (voir figure 1.3a [14]) qui comprend trois régions. La première partie correspond aux défauts de jeunesse intervenant au début de la vie du circuit, principalement dus à des défauts de fabrication. La deuxième partie correspond à des défauts aléatoires qui interviennent

1. Failure In Time : nombre de défaillances par 10^9 h

pendant la période d'utilisation du circuit. Et la troisième partie correspond aux défauts d'usure qui interviennent à la fin de la vie du circuit.

Concernant les différents types d'erreurs mentionnés dans le paragraphe 1.1.1, les erreurs permanentes et intermittentes se retrouvent principalement dans la troisième partie, mais aussi dans la première. Alors que, les erreurs transitoires peuvent intervenir dans chacune des ces parties.

La courbe 1.3a représente l'évolution statistique du taux de défaillance d'un composant, mais il est intéressant d'observer l'évolution correspondante à un ensemble de composants. La figure 1.3b [14] montre cette évolution sur une population de composants semi-conducteurs. Pour une population de composants donnée, la première partie de la période d'utilisation des composants est caractérisée par la défaillance des composants comportant des défauts de fabrication. Ces défaillances peuvent survenir après seulement quelques dizaines d'heures. Vient ensuite la période de vie utile, où seuls quelques composants isolés subissent des défaillances, dues à des conditions d'utilisation aux limites, à un stress excessif ou à l'environnement. Finalement, la fin de vie des composants est caractérisée par une défaillance globale des composants, due au vieillissement des structures internes de ceux-ci.

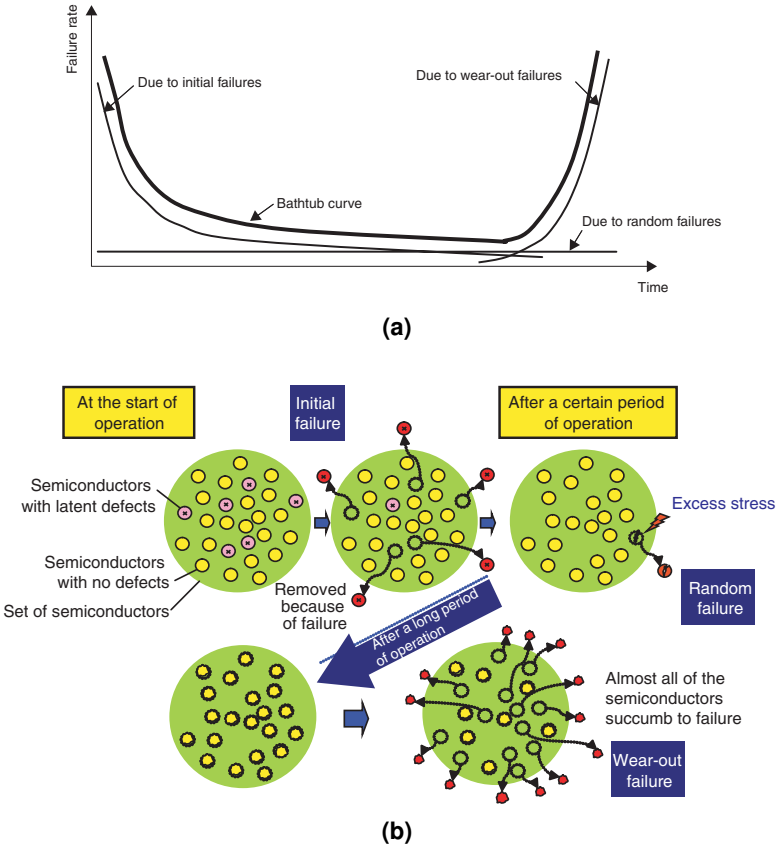


FIGURE 1.3 : *Évolution du taux de défaillance [14] (a) au court de la vie d'un composant (b) appliquée à une population de composants.*

1.1.3 Mesure de la fiabilité d'un système

La fiabilité d'un système est souvent évaluée à partir du *taux de défaillance* mais le *taux de disponibilité* devient une information plus importante. En effet, avec la diminution du facteur d'échelle, le *taux de défaillance* devient de plus en plus difficile à maintenir, contrairement au *taux de disponibilité*. Ainsi, il est possible d'obtenir un système hautement disponible en ayant un système faiblement fiable (voir figure 1.4).

La *disponibilité* ou le *taux de disponibilité D* d'un système, représente son temps de fonctionnement efficace et est défini en fonction de son *temps moyen de fonctionnement avant défaillance* et de son *temps moyen d'indisponibilité*[3].

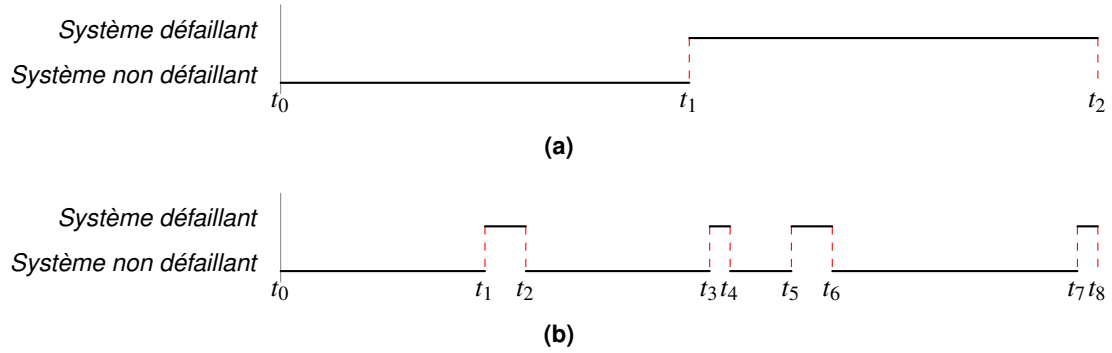


FIGURE 1.4 : *(a) Système fiable mais faiblement disponible. (b) Système faiblement fiable mais disponible.*

Temps moyen de fonctionnement avant défaillance

Pendant la période de vie utile d'un composant, on considère que le taux de défaillance λ est constant et on définit le *temps moyen de fonctionnement avant défaillance*, désigné par le sigle anglais MTTF [2]. Le MTTF est défini à partir de l'inverse du taux de défaillance λ :

$$MTTF = \int_0^\infty e^{-\lambda t} dt = \frac{1}{\lambda} \tag{1.1}$$

Il est admis que le MTTF d'un circuit électronique est compris entre 2 et 20 ans.

Temps moyen de réparation

Dans le cas d'un système réparable il est possible de définir le *temps moyen de réparation* ou *temps moyen d'indisponibilité*, désigné par le sigle anglais MTTR [3]. Il comprend le temps mis pour arrêter le système, le temps de réparation/reconfiguration et le temps de restauration du système.

2. Mean Time To Failure
3. Mean Time To Repair

Pour un système tolérant aux erreurs, ce temps dépend des techniques de détection et de correction/restauration employées. Par exemple, les techniques de masquage impliquent un MTTR très faible. C'est le cas pour un système implémentant des codes correcteurs d'erreurs comme l'ECC [4] pour masquer des erreurs dans les mémoires. Mais ce temps peut considérablement augmenter dans certains cas : s'il est nécessaire de faire une réinitialisation complète du système, de recharger un contexte, etc.

Le MTTR est l'inverse du taux de réparation, généralement noté μ : $MTTR = \frac{1}{\mu}$

Temps moyen entre deux erreurs

A partir des deux grandeurs précédentes on peut définir le *temps moyen entre deux erreurs*, désigné par le sigle anglais MTBF [5] : $MTBF = MTTF + MTTR$.

Pour un système non réparable, c'est à dire que la première défaillance cause la fin de vie du système, on a $MTTR = \infty$, et on utilise alors uniquement le terme $MTTF$.

La figure 1.5 présente une illustration de ces constantes.

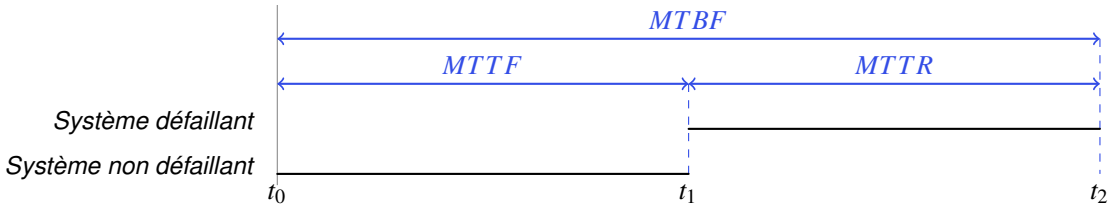


FIGURE 1.5 : *Illustration du temps moyen de fonctionnement avant défaillance (MTTF), temps moyen de réparation (MTTR) et du temps moyen entre deux erreurs (MTBF).*

Calcul de la disponibilité d'un système

On définit alors le taux de disponibilité d'un système par le rapport du *temps moyen de fonctionnement avant défaillance* sur le *temps moyen entre deux erreurs* :

$$D = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF} = \frac{1/\lambda}{1/\lambda + 1/\mu} = \frac{\mu}{\lambda + \mu} \tag{1.2}$$

Le tableau 1.1 [15] donne un exemple de valeurs de disponibilité utilisées pour les systèmes industriels, tels que les serveurs de calcul. On peut constater que le terme de *système tolérant aux fautes* correspond ici à un système ayant un taux de disponibilité d'au moins 99.9999%.

En pratique le taux de disponibilité peut aussi être utilisé pour calculer le temps de réparation minimal ou le temps minimal avant défaillance nécessaire pour atteindre un certain niveau de disponibilité.

4. Error Correction Code
5. Mean Time Between Failures

Prenons un processeur muni d'un *watchdog* [16] pour détecter des erreurs d'exécution. On considère qu'un reset doit être effectué à la détection d'une erreur. Dans ce cas, le $MTTR$ est égal au temps entre la détection de l'erreur et la reprise du système.On désire un taux de disponibilité de $99,99\%$, ce qui équivaut à une moyenne de $52,6$ minutes d'indisponibilité par an ou $8,6$ secondes par jour. Si le processeur a un $MTTF$ de 2 ans, alors le $MTTR$ doit être inférieur à $1,75$ heures. Et à l'inverse si le processeur a un $MTTR$ de 1 seconde alors il peut tolérer un $MTTF$ de $2,8$ heures.

Pourcentage de disponibilité	Temps d'indisponibilité par an	Classification
99,5%	3,7 jours	Conventionnel
99,9%	8,8 heures	Disponible
99,99%	52,6 minutes	Hautement disponible
99,999%	5,3 minutes	Résistant aux erreurs
99,9999%	32 secondes	Tolérant aux erreurs

TABLE 1.1 : *Classification de la disponibilité des systèmes[15]*

1.1.4 Synthèse et positionnement de l'étude

La partie précédente permet de détailler les mesures qui permettent d'estimer la fiabilité d'un système. Ainsi, les deux paramètres principaux sont le MTTF, qui définit la fiabilité d'un système, et le MTTR, qui définit le temps nécessaire au rétablissement d'un système. Le MTTF est donné par la technologie et ne peut être modifié après la fabrication. A la différence du MTTR, qui n'est pas dépendant de la technologie, mais de la conception du système.

Ainsi, le concepteur d'un SoC n'a que peu d'influence sur le MTTF de son circuit, mais peut intervenir sur la disponibilité de son système. Ainsi pour augmenter la disponibilité, il devra faire en sorte de réduire le temps nécessaire à la détection des erreurs et le temps de restauration du système, réduisant ainsi le MTTR. On constate donc que la détection des erreurs est au cœur de la fiabilité et sera un point d'attention tout au long du mémoire.

Nous avons pu voir précédemment qu'il existe trois grandes classes de fautes : permanentes, transitoires et intermittentes. Le système de détection des erreurs doit prendre en compte ces différents types de fautes afin de protéger au mieux le système. Contrairement aux fautes permanentes et transitoires, les fautes intermittentes sont peu étudiées. Or, il est important de connaitre la source de ces fautes pour pouvoir s'en prémunir.

Ainsi, la partie suivante décrit les différentes sources de défaillances dans les circuits intégrés, et fait le lien avec la génération des fautes intermittentes.

1.2 Les sources de défaillances matérielles dans les SoC

Cette partie fait un inventaire des problèmes rencontrés dans les circuits électroniques actuels et permettra de relier ces problèmes aux fautes intermittentes. Les premiers problèmes se rencon-

trent dès la conception des circuits intégrés, se poursuivent dans le temps par des problèmes de vieillissement et sont amplifiés par les variations de la tension d'alimentation et de la température. La combinaison de ces phénomènes est en cause dans l'apparition des fautes intermittentes dans les circuits intégrés.

1.2.1 Fonctionnement du transistor MOS

Avant de s'intéresser aux problèmes de conception et de vieillissement des circuits intégrés, il est nécessaire d'introduire quelques notions de base à propos des transistors. Le transistor est le composant de base utilisé dans les circuits intégrés numériques. Cette partie rappelle brièvement le fonctionnement du transistor et explique comment la modification de sa structure peut avoir un impact sur les délais de propagation.

Constitution du transistor MOS

Le transistor MOS pour Métal Oxyde Semiconducteur est le transistor le plus utilisé. Il est constitué d'une électrode métallique, d'un isolant (oxyde métallique) et d'un substrat semi-conducteur [17]. L'électrode métallique est placée sur un oxyde métallique qui l'isole du substrat semi-conducteur (voir figure 1.6).

Il existe principalement deux types de transistors MOS selon que le substrat est dopé N ou P. On nommera alors NMOS un transistor à canal N et PMOS un transistor à canal P. La suite de ce chapitre prend pour exemple un NMOS, mais le principe de fonctionnement reste identique pour les deux types de transistors.

Le transistor MOS est composé de trois éléments, la Source, le Drain et la Grille. Le Drain et la Source sont connectés à deux régions fortement dopées de nature contraire au substrat. Sur la figure 1.6, pour le cas d'un NMOS, on trouve le Drain et la Source connectés à des régions dopées $N+$ alors que le substrat est dopé P. La Grille est connectée au substrat par l'intermédiaire d'un isolant.

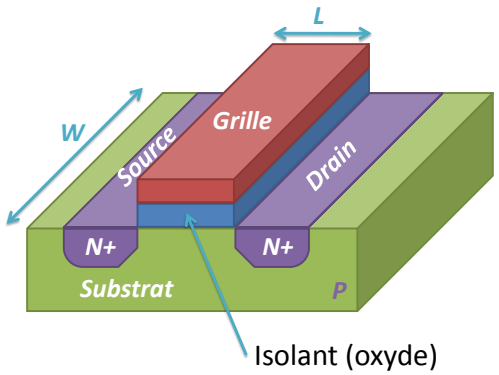


FIGURE 1.6 : *Constitution du transistor MOS. Une électrode métallique est placée sur un oxyde métallique qui l'isole du substrat semi-conducteur. Exemple d'un MOS à canal N (NMOS) avec un substrat dopé P.*

Fonctionnement du transistor en commutation

L'ensemble Grille-Isolant-Substrat fonctionne à la manière d'un condensateur. On applique le potentiel le plus faible au niveau du substrat et une tension positive au niveau de la Grille. Ceci a pour effet d'attirer des charges négatives au niveau du substrat entre la Source et le Drain (voir figure 1.6), ce qui constitue un canal de conduction.

La vitesse de création du canal est proportionnelle au temps de charge du condensateur. Plus la capacité est faible et plus sa charge est rapide. La capacité C_{ox}formée par l'oxyde est fonction de sa surface S , de sa constante diélectrique ε_{ox} et de son épaisseur T_{ox}, la formule est la suivante :

$$C_{ox} = \frac{\varepsilon_{ox} S}{T_{ox}} \tag{1.3}$$

La Source est généralement connectée au substrat et l'attention est portée sur la tension Grille-Source V_{GS}. La modulation de cette tension permet de modifier la résistivité du canal.

- Si V_{GS} est supérieure à la tension de seuil[6] V_{th} alors la résistance est faible, le canal est fermé et le transistor fonctionne comme un interrupteur fermé. Dans ce cas on dit que le transistor est passant.
- Si V_{GS} est nulle alors la résistance est importante, le canal est ouvert et le transistor fonctionne comme un interrupteur ouvert. Dans ce cas on dit que le transistor est bloqué.

Ce fonctionnement permet d'utiliser le transistor en commutation en électronique numérique. Si V_{GS} est supérieure à V_{th} alors le niveau logique est haut ("1"), sinon le niveau logique est bas ("0").

La tension de seuil dépend directement des paramètres technologiques, comme le dopage du substrat ou les défauts dans l'isolant et de la température [17].

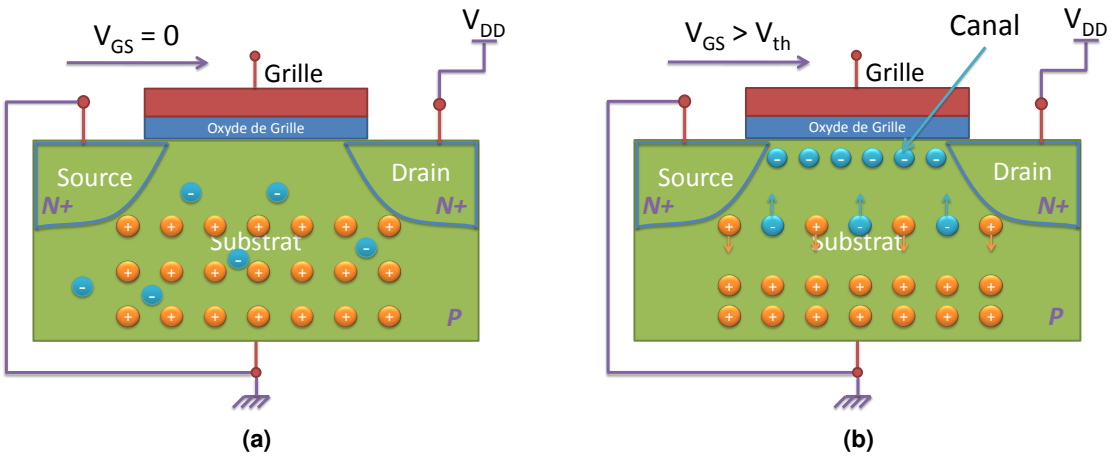


FIGURE 1.7 : *Fonctionnement du MOS en commutation, dans le cas d'un MOS à canal N (NMOS). (b) Fonctionnement en interrupteur ouvert :* $V_{GS} = 0 \Rightarrow$ *canal fermé. (a) Fonctionnement en interrupteur fermé :* $V_{GS} > V_{th} \Rightarrow$ *canal ouvert.*

6. en anglais Threshold Voltage

Fréquence de fonctionnement

Afin de s'intéresser à la fréquence de fonctionnement d'un circuit constitué de transistor MOS, prenons pour exemple l'inverseur. L'inverseur est un des circuits les plus simples, il est constitué d'un transistor NMOS et d'un transistor PMOS. Le schéma de l'inverseur est présenté sur la figure 1.8, la capacité C_L représente la capacité de charge du circuit suivant l'inverseur. Voici son fonctionnement :

- Si l'entrée V_{in} est attaquée par un niveau logique haut, alors le transistor NMOS devient passant et la sortie V_{out} est connectée à la masse, ce qui impose un niveau logique bas.
- Si l'entrée V_{in} est attaquée par un niveau logique bas, alors le transistor PMOS devient passant et la sortie V_{out} est connectée à V_{DD}, ce qui impose un niveau logique haut.

La fréquence de fonctionnement de l'inverseur est définie par le temps de basculement entre le niveau logique haut et le niveau logique bas en sortie. Si l'on appelle τ ce délai de propagation, alors son équation est la suivante [17] :

$$\tau = C_L V_{DD} \frac{2L}{W \mu C_{ox}} \frac{1}{(-V_{DD} - V_{th})^2} \tag{1.4}$$

Dans cette équation, W et L représentent les dimensions du transistor, μ la mobilité des porteurs de charge (mobilité des électrons dans le cas d'un NMOS), C_{ox} la capacité formé par l'oxyde de grille, V_{DD} est la tension d'alimentation et V_{th} la tension de seuil du transistor.

On considère que les dimensions du transistor et la capacité de charge du circuit en sortie de l'inverseur ne varient pas après fabrication. Ainsi, en fonctionnement, le délai de propagation peut varier en fonction de la capacité formé par l'oxyde de grille (C_{ox}), de la tension d'alimentation (V_{DD}) et de la tension de seuil du transistor (V_{th}).

La moindre modification d'un de ces trois paramètres influe sur le temps de propagation. Nous verrons dans la suite de ce chapitre comment ces paramètres peuvent être affectés par les variations des procédés de fabrication, de la température et du vieillissement.

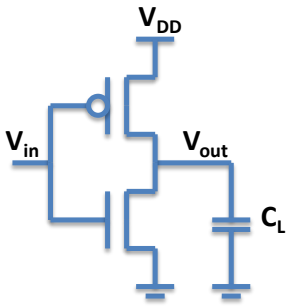


FIGURE 1.8 : *Schéma de l'inverseur, constitué d'un NMOS (en bas) et d'un PMOS (en haut).*

1.2.2 Les variations de paramètres technologiques dans les circuits intégrés

Le premier problème auquel sont confrontés les circuits intégrés actuels est le problème des variations PVT [7]. Les variations PVT désignent les variations de paramètres technologiques dues à des variations du procédé de fabrication (*Process*), de la tension d'alimentation (*Voltage*) et de la température (*Temperature*). Ces problèmes ne sont pas nouveaux, mais ils tendent à s'amplifier dans les technologies actuelles et à venir.

Avec la diminution du facteur d'échelle, les procédés de fabrication n'induisent plus seulement des variations des paramètres entre les composants d'un même wafer [9] (die-to-die), mais entre les transistors d'un même composant [18–20] (within-die). Ces variations ont un impact direct sur la taille des chemins critiques et donc sur la fréquence maximale de fonctionnement de chacun des circuits intégrés réalisés. Ainsi, d'une part, deux composants d'un même lot peuvent avoir deux fréquences maximales de fonctionnement différentes. Et d'autre part, deux parties d'une même puce peuvent aussi avoir deux fréquences maximales de fonctionnement différentes. La partie du circuit intégré fonctionnant ayant la fréquence de fonctionnement maximale la plus faible sera donc plus sensible à des erreurs de délais.

La diminution du facteur d'échelle permet de diminuer la consommation des circuits mais impose des tensions d'alimentation difficiles à atteindre. De même, de grandes différences de température peuvent apparaitre entre les différents modules d'un même composant. Ce qui a pour conséquence des problèmes de fiabilité et de performances.

Variations du procédé de fabrication

Les variations causées par les procédés de fabrication apparaissent à deux niveaux : entre les puces d'un même wafer (D2D [8]) et au sein d'une même puce (WID [9]). Ces variations ont un impact sur la fréquence maximale de fonctionnement des composants ainsi que sur les courants de fuite des transistors. La figure 1.9 illustre les deux phénomènes.

Au vu de la taille des transistors dans les technologies actuelles, les étapes de fabrication doivent être de plus en plus précises. Cependant, sans entrer dans le détail des procédés industriels, une dérive sur la taille du canal des transistors ou sur le dopage des semi-conducteurs peut avoir des conséquences importantes sur le fonctionnement de la puce (voir § 1.2.1). En particulier, cela influe sur la fréquence maximale de fonctionnement et le courant de fuite des circuits intégrés. La figure 1.10 met en évidence les variations des fuites du courant de repos (I_{sb}) et de la fréquence dues aux variations de procédé de fabrication [9], pour un ensemble de processeurs d'un même wafer (variations D2D). Borkar et al. montrent sur cette figure, qu'il peut y avoir des dispersions de l'ordre de 30% de la fréquence maximale de fonctionnement des processeurs et des dispersions de l'ordre de 2000% en ce qui concerne les fuites de courant. Ces valeurs sont

7. Process Voltage Temperature
8. die-to-die
9. within-die

données pour une technologie 180nm et ont tendance à augmenter avec la diminution du facteur d'échelle.

Concernant la distribution statistique de la fréquence maximale de fonctionnement, Bowman et al. [19] ont conclu que les variations WID ont un impact sur la moyenne et que les variations D2D ont un impact sur la variance. La diminution de la fréquence maximale de fonctionnement due aux variations WID serait de l'ordre de 40% pour une technologie 50nm. En comparaison la diminution de la fréquence maximale était de l'ordre de 25% pour une technologie 180nm.

Au final, pour un composant donné dans une technologie actuelle, il est rare que tous les transistors et toutes les interconnexions aient les mêmes paramètres technologiques. Ceci conduit à des vitesses de fonctionnement différentes au sein du composant. Si la différence des fréquences de fonctionnement est trop importante cela peut conduire à des problèmes de synchronisation. De plus les parties ayant la fréquence de fonctionnement maximale la plus faible seront plus sensibles à des fautes de délais. Cela peut conduire à des erreurs intermittentes.

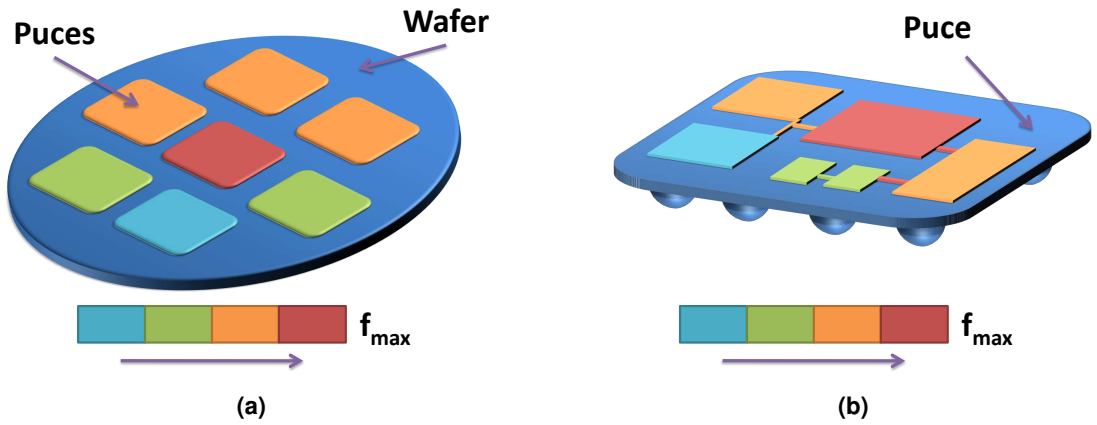


FIGURE 1.9 : *Illustration de l'impact des variations de procédés de fabrication sur les fréquences maximales de fonctionnement. (a) Variations entre les puces d'un même wafer (D2D). (b) Variations entre les différentes parties d'une même puce (WID).*

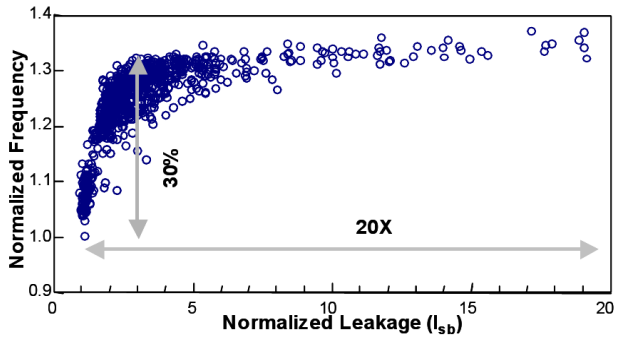


FIGURE 1.10 : *Variations du courant de fuite et de la fréquence dues aux variations D2D [9]. Les données sont normalisées à partir de la plus faible valeur de fréquence et de la plus faible valeur de courant de fuite.*

Variations de la tension d'alimentation

Vis à vis de fiabilité, une mauvaise valeur de tension peut avoir un impact négatif, si elle est supérieure à une tension maximale V_{max} ou si elle est inférieure à une tension minimale V_{min} [9] (voir figure 1.11).

La tension V_{min} définit un seuil en dessous duquel les performances sont impactées. En effet, comme nous l'avons vu au paragraphe 1.2.1, plus la tension d'alimentation est faible et plus le temps de propagation est important.

La tension V_{max} est imposée par la technologie et les contraintes de consommation. Si la tension est trop importante, d'une part cela augmente la consommation du circuit, ce qui est critique dans les systèmes embarqués. D'autre part, cela augmente la température, ce qui est aussi néfaste pour la fiabilité (voir paragraphe suivant).

Premièrement, l'activité des différentes parties du composant provoque des variations de la tension d'alimentation, ce qui peut conduire à des pics de tension (voir figure 1.11). Ces variations temporaires, peuvent induire des violations de délai, ce qui peut conduire à des erreurs intermittentes.

Deuxièmement, la tension d'alimentation n'est pas abaissée en respectant le facteur d'échelle. Historiquement la tension d'alimentation diminuait de 30% d'une génération de composants à une autre, mais actuellement cette diminution s'approche des 15% [9]. Ainsi, la tension d'alimentation tend à être trop importante. Srinivasan et al. [4] ont montrés à partir d'une simulation de vieillissement d'un processeur en 65nm, que la vitesse de dégradation du composant peut être multipliée par deux si la tension d'alimentation est de 1V au lieu de 0,9V.

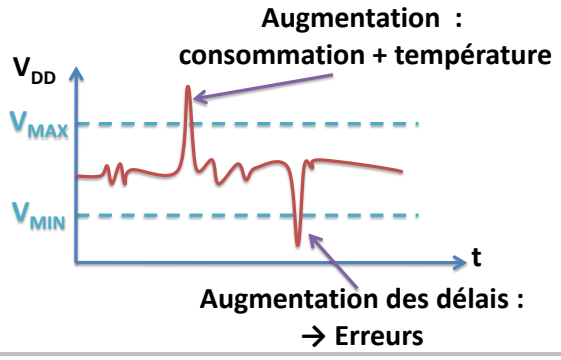


FIGURE 1.11 : *Illustration de l'impact des variations de la tension d'alimentation.*

Variations de la température

La température joue également un rôle important sur les délais de propagation et la performance des composants, plus la température est importante et plus la dégradation des performances est grande. La figure 1.12 illustre le phénomène.

Au niveau des transistors, c'est la tension de seuil (V_{th}) qui en est affectée. À une température de 100°C la tension de seuil est abaissée de 20mV par rapport à une température de 25°C [21]. Or, comme nous l'avons vu au paragraphe 1.2.1, le délai de propagation est directement lié à la tension de seuil.

La figure 1.13 représente la répartition de la température d'un processeur en cours de fonctionnement [9] dans une technologie 180nm. Cette figure met en évidence des différences de température de plus de 50°C à l'intérieur du composant, qui évoluent avec le temps. De telles variations de la température induisent des variations non uniformes des délais à l'intérieur du composant. De plus, de telles températures amènent le composant à se détériorer plus rapidement (voir § 1.2.3).

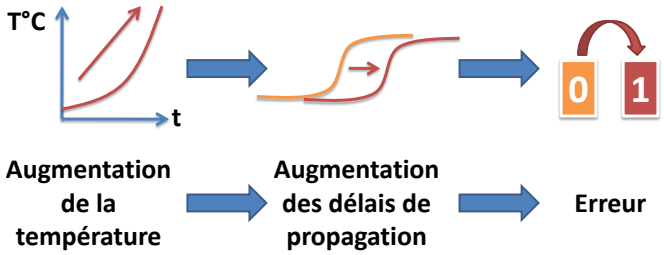


FIGURE 1.12 : *Illustration de l'impact des variations de la température.*

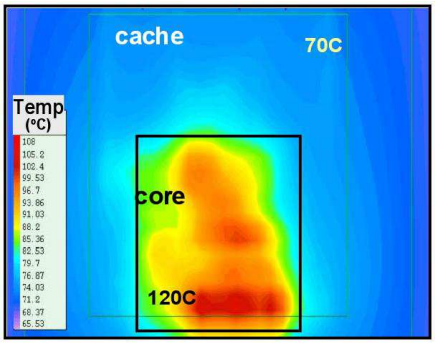


FIGURE 1.13 : *Variation de la température à l'intérieur d'une puce. [9]*

Combinaison des variations procédé-température-tension

Tout composant est soumis aux variations précédentes et au final, c'est une combinaison de toutes ces variations qui est source d'erreurs. En effet, les délais de propagation sont affectés dans chacun des cas, et ont un impact direct sur les marges temporelles nécessaires au bon fonctionnement des composants. Les variations au niveau des procédés de fabrication peuvent dégrader les performances de circuits ou de parties de circuits. Ils mettent ainsi en évidence des circuits faibles qui sont plus sensibles aux fautes de délais. Si l'on ajoute des variations de la tension

d'alimentation et de la température, alors ce sont ces circuits faibles qui ont la grande probabilité de générer des erreurs.

Afin de minimiser ces phénomènes, les fabricants de circuits intégrés augmentent les marges temporelles. Cependant, si les marges sont sur-évaluées le coût de conception pour atteindre les performances attendues va augmenter. Et si les marges sont sous-évaluées c'est le rendement qui est impacté, ce qui induit une augmentation des coûts de production. La figure 1.14 montre l'impact de chacune des variations de procédé, de température et de tension sur les marges temporelles pour un procédé en 65nm, une tension minimale de 1.15 volts et une température nominale de 80°C [22]. On peut observer que les marges nécessaires à appliquer sont supérieures à 30%.

Ces marges temporelles sont calculées de manière à pallier aux variations maximales sur le chemin critique. Or, ces variations peuvent affecter des chemins logiques secondaires, créant de nouveaux chemins critiques et rendant ainsi difficile la détection des erreurs. De plus, il est difficile de prévoir un changement des conditions de fonctionnement extérieures ou le vieillissement du système qui peuvent amener des variations des marges temporelles au court du temps.

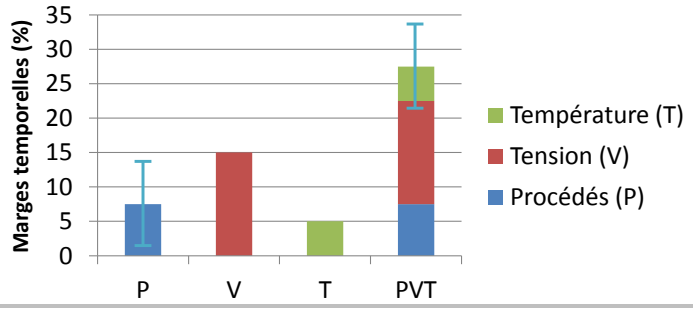


FIGURE 1.14 : *Impact des variations PVT sur les marges temporelles [22]. Exemple de marges temporelles pour un procédé en 65nm, une tension minimale de 1,15V et une température nominale de 80 °C*

1.2.3 Les défaillances matérielles dans les circuits intégrés

Au cours de la vie d'un circuit, chacune des parties qui le compose va s'user. Cette usure va dépendre de plusieurs paramètres : le design de la puce, les tensions d'alimentation, les courants en jeu, les matériaux sélectionnés, la taille et la forme des pistes, etc. L'usure dépend aussi de l'utilisation du circuit et des conditions extérieures (température, taux d'humidité, etc). Pour finir, cette usure va évoluer en fonction du temps.

Le vieillissement est caractérisé par différents mécanismes de défaillance qui agissent indépendamment en fonction des paramètres cités précédemment et sont localisés sur des parties spécifiques du circuit. Par exemple, le phénomène d'électromigration est un mécanisme de défaillance qui touche principalement les lignes métalliques du circuit et qui évolue en fonction du matériau utilisé, de la densité de courant, de la température et du temps (voir § 1.2.3).

Il existe beaucoup de mécanismes de défaillances qui touchent la puce [17] mais aussi le boitier. Par exemple, en fonction des stress mécaniques appliqués sur le composant, des défauts

peuvent apparaître sur les contacts entre le composant et son support, mais ces mécanismes ne sont pas le sujet de ce document. Nous allons nous concentrer principalement sur ceux qui touchent les interconnexions et les oxydes de grille des transistors. Cette section présente les phénomènes d'*Électromigration* et de *Metal Stress Voiding* qui sont les principaux acteurs de la dégradation des lignes de métal, ainsi que les phénomènes de *Time Dependent Dielectric Breakdown*, de *Hot Carrier Injection* et de *Negative Bias Temperature Instability*, qui eux dégradent l'oxyde de grille des transistors.

Dégradation des lignes de métal

Électromigration L'électromigration (EM) est un mécanisme de défaillance qui dégrade les lignes de métal et qui peut aller jusqu'à causer des ruptures de lignes ou des jonctions de plusieurs lignes [23] (voir figures 1.15a et 1.15b[24]). L'électromigration peut ainsi causer des circuits-ouverts ou des court-circuits.

Le principe de fonctionnement de l'électromigration est schématisé sur la figure 1.16. Ce phénomène est causé par un flux d'électrons et est aggravé par la température. Les électrons, en se déplaçant dans le matériau, entrent en collision avec des atomes de métal. Si l'énergie communiquée aux atomes de métal lors de cette collision est suffisamment importante, ces atomes se déplacent dans le sens du flux d'électrons. La zone appauvrie devient un vide, qui peut conduire à un circuit ouvert et la zone en surplus peut causer des courts-circuits avec les lignes voisines. Ce phénomène affecte surtout les longues lignes de métal dans lesquelles le courant est continu et ne circule que dans un sens.

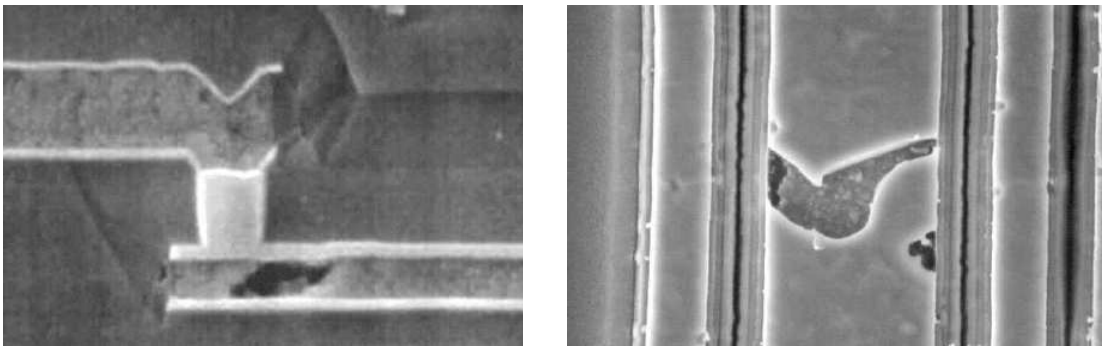
Avant de provoquer la rupture totale d'une ligne, un "creux" est créé. Ce creux diminue la surface de la section de la ligne localement et augmente ainsi sa résistance. Ce qui a deux effets : d'une part, l'augmentation de la résistance augmente les délais sur la piste, ce qui peut créer une erreur de timing ; d'autre part, cela augmente la température localement par effet Joule, puis la densité de courant, ce qui amplifie le phénomène.

L'évolution des technologies implique une diminution du facteur d'échelle des composants intégrés. A ce titre, la section des lignes de métal n'est pas épargnée. Plus la technologie évolue et plus la densité de courant augmente, accélérant l'apparition du phénomène. Cependant, ce phénomène peut être réduit par l'ajout de couches de Tungsten ou de Titane ou l'ajout de particules de Cuivre dans le cas d'une ligne d'Aluminium.

Le modèle généralement accepté pour décrire le MTTF [10] lié à l'EM est donné dans le tableau de synthèse 1.2 page 26.

Metal Stress Voiding Le *Metal Stress Voiding* (MSV) ou *Stress Voiding* ou encore *Stress Migration* est un mécanisme de défaillance qui cause des dégradations sensiblement identiques à l'EM (voir figures 1.17a et 1.17b), à la différence que les atomes de métal ne se déplacent pas à cause d'un flux d'électrons mais à cause de contraintes mécaniques [24]. Plus particulièrement à

10. Mean time-to-failure



(a) *Coupe d'un via dégradé par EM* **(b)** *Interconnexion dégradée par EM*

FIGURE 1.15 : *Exemples de dégradations par Électromigration [24]*

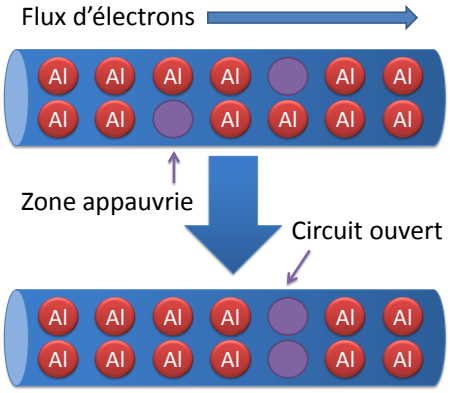


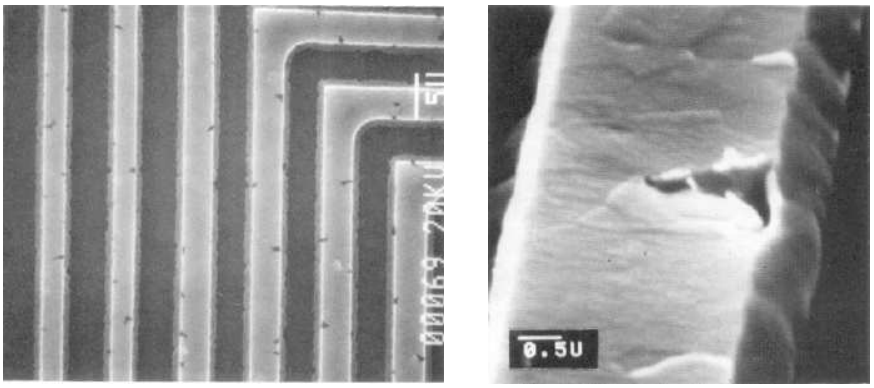
FIGURE 1.16 : *Principe de l'Électromigration. Dans certaines conditions, les atomes se déplacent dans le sens du flux d'électrons. La zone appauvrie devient un vide, qui peut conduire à un circuit ouvert.*

cause des différences de taux d'expansion entre les lignes de métal et la couche de passivation. Ainsi, il n'est pas nécessaire qu'un courant circule pour que le phénomène se déclare.

Lors de la fabrication du composant, la température des lignes de métal peut atteindre 400°C ou plus lors de l'étape de passivation, le métal s'étend et s'accroche fermement à la couche de passivation. Mais quand le circuit revient à la température ambiante, des tensions importantes se créent dans le métal à cause de la différence de coefficient thermique d'expansion des deux matériaux.

Comme dans le cas de l'électromigration, la piste partiellement coupée accroît localement la résistance à cause de la diminution de la section. Les délais de propagation sont donc augmentés. De plus, la diminution de la section fait augmenter la densité de courant, ce qui a pour conséquence de favoriser le phénomène d'électromigration.

Le modèle généralement accepté pour décrire le MTTF lié au MSV est donné dans le tableau de synthèse 1.2 page 26.



(a) *Trous sur lignes de métal* **(b)** *Dégradation d'une ligne de métal*

FIGURE 1.17 : *Dégradations par MSV[24]*

Dégradation des oxydes de grille

Time-Dependent Dielectric Breakdown Le *Time-Dependent Dielectric Breakdown* (TDDB) ou *Gate Oxyde Breakdown* ou *Gate Oxyde Wearout* ou encore claquage d'oxyde est un phénomène qui affecte la couche isolante de dioxyde de silicium au niveau de la grille. Il est dû au claquage de la couche d'oxyde de la grille qui peut s'apparenter au claquage d'un condensateur.

Lorsque les électrons ont réussi à franchir la barrière de potentiel de l'oxyde, ils sont accélérés par le champ électrique présent entre la grille et le substrat. Ils acquièrent alors une forte énergie qui est libérée à l'interface oxyde/silicium [23]. Il existe entre l'oxyde et le silicium une grande différence de coefficient de dilatation qui entraîne des tensions mécaniques sur les liens de cette interface. Ces liens peuvent être coupés à cause de l'énergie libérée par les électrons accélérés. Cela conduit à la création de zones de "piégeage". Les charges piégées diminuent la mobilité des porteurs et ralentissent donc le transistor. Ces retards peuvent abaisser la fréquence maximale de commutation du transistor et ainsi diminuer le chemin critique du composant. Les charges piégées augmentent également le champ électrique localement, ce qui augmente le courant par effet tunnel à travers l'oxyde.

Cet effet peut donc s'amplifier et conduire à la création d'un chemin de conduction (voir figure 1.18). Une fois créé, ce chemin conduit à un courant trop fort dans la couche d'oxyde. Ce qui augmente la température de la grille et conduit à sa destruction.

De même que pour les phénomènes précédents, avant la destruction du transistor, on peut observer une augmentation des délais de propagation qui peuvent conduire à des erreurs intermittentes.

Le modèle généralement accepté pour décrire le MTTF lié au TDDB est donné dans le tableau de synthèse 1.2 page 26.

Hot Carrier Injection Le phénomène "Hot Carrier Injection" (HCI) est dû à l'ionisation causée par l'impact des électrons sur les atomes de silicium au niveau du drain. L'ionisation engendre

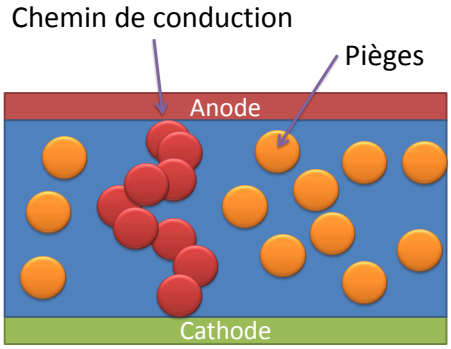


FIGURE 1.18 : *Principe du claquage d'oxyde basé sur la théorie de percolation. Dans le cas du MOS, l'Anode correspond à la Grille métallique, la Cathode au substrat et la partie intermédiaire à l'oxyde.*

une paire électron-trou qui entre dans le substrat et provoque l'augmentation du courant dans le substrat. Une partie des porteurs créés peut alors franchir la barrière de potentiel de la couche d'oxyde de la grille. Le HCI diminue la mobilité des porteurs de charge ce qui augmente les temps de commutation. Les retards, dans le cas où ils apparaissent sur le chemin critique, font baisser la fréquence maximale de commutation du transistor. Ce phénomène est plus important à basse qu'à haute température car les électrons sont plus mobiles et donc leur énergie plus grande lors de l'ionisation.

Ce phénomène peut être évalué en mesurant le courant de saturation du drain I_{Dsat} qui est un des paramètres influant sur la rapidité d'un transistor [23]. Les dommages causés par HCI sur l'oxyde de la grille augmentent la tension de seuil du transistor NMOS ce qui fait décroitre le courant I_{Dsat}.

Le courant circulant dans le canal est maximum lors des commutations, c'est à ce moment que le phénomène HCI est maximal. L'HCI est donc une défaillance qui intervient lorsqu'un processeur est actif.

Il est important de remarquer que ce phénomène n'est pas destructif : la structure du circuit n'est pas modifiée, il peut donc être régénéré [25].

La formule suivante décrit la durée de vie t d'un transistor affecté par un phénomène de HCI [14] (avec A et B des constantes dépendantes de la technologie et de la température). C'est une formule simple qui ne prend en compte que la tension drain-source, mais qui montre que la tension drain-source V_{ds} est le principal facteur aggravant du HCI.

$$t = A.e^{\frac{-B}{V_{ds}}}$$

Negative Bias Temperature Instability Le phénomène *Negative Bias Temperature Instability* (NBTI) est propre au transistor PMOS. Il se produit lorsque des trous à haute énergie bombardent l'interface oxyde/substrat, ils réagissent avec l'oxyde pour produire des atomes d'hydrogène en cassant la liaison Si–H [23]. Ceci crée des zones de charges positives au niveau de l'interface oxyde/substrat ce qui réduit la mobilité des trous et modifie la tension de seuil du transistor. Le courant de drain diminue et les délais augmentent, ce qui abaisse les marges temporelles. Il est a

noter que le même phénomène affecte les transistors NMOS et est appelé Bias Temperature Instability. Ces deux phénomènes sont considérés comme étant prépondérants dans les technologies actuelles [26].

Le MTTF est dépendant de la température et de la tension d'alimentation. Cependant, ici, c'est la tension d'alimentation qui est prépondérante [14]. Le modèle généralement accepté pour décrire le MTTF lié au NBTI est donné dans le tableau de synthèse 1.2 page 26.

Évolution du vieillissement en fonction de la tension et de la température

Il est important de remarquer que tous les phénomènes de dégradation cités précédemment sont fortement liés à la température. Le tableau 1.2 présente un résumé des équations de dégradation relatif aux mécanismes de défaillance précédents, où l'on peut observer le facteur $e^{\frac{E_a}{kT}}$, principal contributeur du vieillissement.

Pour mettre en évidence l'évolution du MTTF avec la température, il suffit de calculer le rapport du MTTF en température par rapport au MTTF à température ambiante, on obtient alors un facteur d'accélération en température. Et on a :

$$A_T = MTTF_{\text{ambiant}}/MTTF_{\text{température}}$$

Si on applique cette formule au MTTF correspondant à l'électromigration entre 50°C et 80°C, on trouve que la dégradation est accélérée par un facteur $15 : 1,5$ fois par la densité de courant et 10 fois par la température (voir détail sur la figure 1.19).

La figure 1.20 présente l'évolution du facteur d'accélération par électromigration pour une ligne Al-Cu longue avec $E_a = 0.8eV$ et considérant une densité de courant invariante avec la température. L'exemple précédent considère une température de 80°C mais nous avons vu dans le § 1.2.2 que cette température peut être supérieure à 120°C, l'accélération du vieillissement devient supérieure à 100.

Les exemples précédents correspondent à l'électromigration mais les mêmes constats pourraient être faits à partir des autres mécanismes de défaillance. Ceci montre qu'il est essentiel de limiter au maximum la température des circuits intégrés afin d'éviter d'accélérer le vieillissement. Néanmoins, cette dépendance à la température est utilisée dans les procédés de tests accélérés. Ces tests permettent d'étudier, à des échelles de temps réduites, des défauts de vieillissement qui apparaissent normalement en fin de vie des circuits intégrés. Pour effectuer ces tests, la température est augmentée aux limites de fonctionnement des circuits pour une durée allant de 200 à 1000 heures [27].

Mécanisme de défaillance	MTTF (h)
EM	$J^{-N}e^{\frac{E_a}{kT}}$
MSV	$(T_0-T)^{-N}e^{\frac{E_a}{kT}}$
TDDB	$(\frac{1}{V})^{a-bT}e^{\frac{X+\frac{Y}{T}+ZT}{kT}}$
NBTI	$A.10^{-\beta E}e^{\frac{E_a}{kT}}$

Avec :
J la densité de courant dans les interconnexions (A/cm^2)
E_a l'énergie d'activation relative au mécanisme de défaillance (eV)
k la constante de Boltzmann
N une constante qui dépend du matériau utilisé pour les interconnexions.
a, b, X, Y, Z des paramètres d'échelle
V la tension
E champ électrique (MV/cm)
β constante dépendant de la technologie
T la température absolue (K)
T_0 la température de dépôt du métal(K)

TABLE 1.2 : *Résumé des équations de dégradation associées aux mécanismes de défaillances : le facteur $e^{\frac{E_a}{kT}}$ est le principal contributeur du vieillissement*

$$\begin{aligned} A_T &= \frac{MTTF_{80°C}}{MTTF_{50°C}} \\ &= \left(\frac{J_{80°C}}{J_{50°C}}\right)^{-2}.exp\left(\frac{E_a}{k}.(\frac{1}{T_{80°C}}-\frac{1}{T_{50°C}})\right) \\ &= \left(\frac{2,0}{2,5}\right)^{-2}.exp\left(\frac{0,8}{8,62.10^{-5}}.(\frac{1}{273+50}-\frac{1}{273+80})\right) \\ &= 1,5\ .\ 10 = 15 \end{aligned}$$

FIGURE 1.19 : *Calcul du facteur d'accélération dans le cas de l'électromigration pour une augmentation de 30 °C. La dégradation est accélérée par un facteur $15 : 1,5$ fois par la densité de courant et 10 fois par la température.*

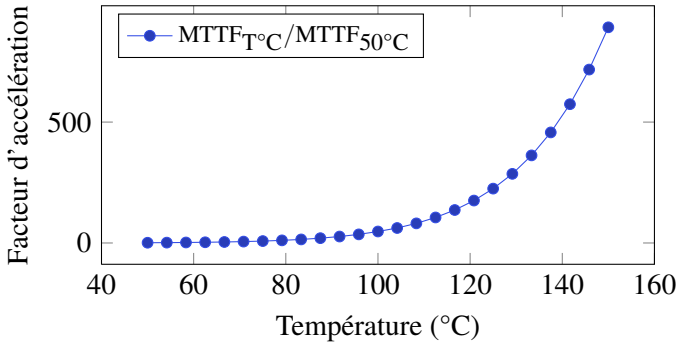


FIGURE 1.20 : *Accélération du vieillissement par électromigration pour une ligne Al-Cu longue avec $E_a = 0.8eV$ et densité de courant invariante.*

1.2.4 Synthèse des problèmes de fiabilité dans les systèmes sur puce et impact sur les erreurs intermittentes

Cette partie fait une synthèse des problèmes de fiabilité affectant les systèmes sur puce actuels et responsables de l'apparition d'erreurs intermittentes.

Les problèmes de fiabilité débutent dès la conception des circuits intégrés par les variations des procédés de fabrication. Ces aléas de fabrication peuvent dégrader les performances de circuits ou de parties de circuits. Ils mettent ainsi en évidence des circuits faibles qui sont plus sensibles aux fautes de délais.

Puis, au court de la vie du circuit, le vieillissement dégrade les lignes de métal et les oxydes de grilles. Cela a pour effet de réduire les marges temporelles, ce qui amplifie la sensibilité des circuits intégrés faibles, mis en évidence par les aléas de fabrication.

De plus, si des variations de la tension ou de la température apparaissent, des erreurs intermittentes peuvent apparaitre. A ce stade, le circuit n'est pas considéré comme défaillant et ces erreurs apparaissent ponctuellement.

Avec le temps et le vieillissement, ces erreurs peuvent devenir permanentes. Dans ce cas, le circuit doit être considéré comme défaillant.

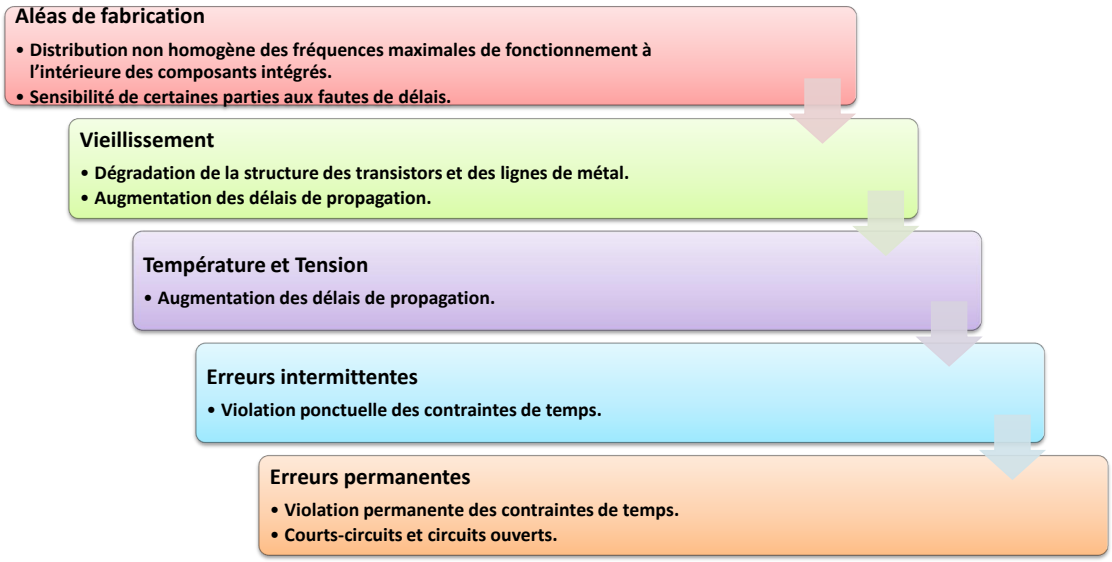


FIGURE 1.21 : *Synthèse des problèmes de fiabilité dans les circuits intégrés et lien avec la génération d'erreurs intermittentes.*

1.3 Conclusion

Ce chapitre a permis de définir, dans un premier temps, les principaux concepts de fiabilité qui seront utilisés au cours de ce mémoire, et dans un deuxième temps, les problèmes de fiabilité affectant les systèmes sur puce. En particulier, nous avons vu que plus la technologie évolue et plus il est difficile de fabriquer des composants sans aucun défaut. Ces défauts ne sont pas forcément visibles dès le début de la vie du composant, et peuvent se matérialiser par des fautes de délais avec le vieillissement du circuit, des fluctuations des tensions d'alimentation ou de la température. Nous avons ainsi montré que, avant que le système ne soit défaillant - pendant sa période de vie utile - des fautes intermittentes peuvent se manifester.

Ne pouvant pas agir directement sur la fiabilité de son système, nous avons montrés que le concepteur peut néanmoins agir sur sa disponibilité. Il doit faire en sorte de réduire le temps nécessaire à la détection des erreurs et le temps de restauration du système. Trois classes de fautes sont a prendre en compte, les fautes permanentes, transitoires et intermittentes. Les deux premières classes sont bien connues et de nombreuses méthodes existent pour les détecter, mais ce n'est pas le cas pour les fautes intermittentes.

Or, les fautes intermittentes sont au cœur de ce mémoire. Une étude approfondie de ces fautes doit être réalisée afin d'adapter au mieux le système de détection. Ce sera l'objectif du prochain chapitre. Les chapitres suivants feront un état de l'art des méthodes de tolérances aux fautes, ce qui permettra de poser les bases de notre solution.

CHAPITRE 2

Mise en évidence des erreurs intermittentes et de leur impact sur les systèmes embarqués

Sommaire

LE CHAPITRE précédent a permis de se rendre compte des problèmes auxquels sont soumis les circuits intégrés au cours du temps. En particulier, les variations du procédé de fabrication associées à la dégradation du circuit, combinées à des fluctuations de la température et de la tension d'alimentation, peuvent être une source importante de fautes intermittentes.

Or, pour tenter de se prémunir de ces fautes, il est important de comprendre leur comportement, ainsi que leur impact sur le système et les applications. Nous allons pour cela réaliser une étude expérimentale pour comprendre comment détecter, voire anticiper l'apparition des fautes intermittentes.

Plus précisément, nous allons pouvoir déterminer quels sont les facteurs prépondérants dans l'apparition des fautes intermittentes. La question se pose en particulier pour l'activité des processeurs : est-ce que deux processeurs soumis à des activités différentes ont un taux d'erreurs intermittentes équivalent ? De plus, si ces fautes ne sont pas destructives, le système peut-il être rétabli ? Les réponses à ces questions vont permettre de déterminer et d'adapter au mieux un système de détection des erreurs intermittentes à une architecture multiprocesseur. Comme nous l'avons vu au chapitre 1, une *erreur* est la manifestation d'une *faute*. Ainsi, dans ce chapitre, nous parlerons de détection des *erreurs*. En effet, l'étude expérimentale nous permettra d'observer des erreurs à partir d'applications exécutées sur des processeurs. Les moyens mis en œuvre ne permettent pas d'observer directement les fautes intermittentes.

La première partie de ce chapitre présentera les objectifs et la méthodologie utilisée. La deuxième partie présentera le principe des tests accélérés qui sera utilisé afin d'accélérer l'apparition des erreurs intermittentes dans notre étude. Les parties trois et quatre décriront la plateforme expérimentale et le protocole expérimental. La partie cinq détaillera les résultats obtenus et mettra en valeur les résultats pouvant être utilisés pour le développement d'une solution de test en ligne des erreurs intermittentes.

Nous montrerons qu'il est possible de détecter des erreurs intermittentes à un niveau système, que le taux d'erreur dépend de l'activité des processeurs, mais que l'impact des erreurs n'est pas le même suivant les applications exécutées. De plus, nous montrerons que les erreurs intermittentes surviennent en rafales et que l'arrêt du système peut suffire à le régénérer.

2.1 Objectifs et méthodologie

L'étude présentée dans ce chapitre s'inscrit dans le domaine multiprocesseur et va tenter de répondre aux questions liées au test en ligne de l'architecture. Premièrement, cette étude doit nous permettre d'étudier le lien entre l'activité des composants et leur vieillissement. Deuxièmement, elle doit nous permettre d'observer des erreurs intermittentes et leur impact sur les applications. Les paragraphes suivants détaillent les deux points précédents et concluent sur la méthodologie adoptée pour répondre à nos objectifs.

Lien entre activité des composants et vieillissement

À la vue des différents mécanismes de vieillissement présentés dans le chapitre 1, il est intéressant de relier le vieillissement des composants avec leur activité. Au niveau du transistor certains mécanismes de défaillance s'activent soit lors des commutations (cas de EM, SM et HCI), soit lorsque le transistor est au repos (cas de TDDB et NBTI). Ainsi, pour un même composant soumis à des activités différentes, la vitesse de dégradation peut être différente. La question se

pose en particulier pour un système multiprocesseur. Si les différents processeurs de l'architecture ne sont pas utilisés de la même façon au cours du temps, alors les vitesses de vieillissement peuvent être différentes. Cependant, il n'est pas possible de prédire, pour une technologie donnée, ni quel composant vieillira le plus rapidement, ni quelle sera l'activité de chaque composant. Cela dépend de la technologie utilisée ainsi que des conditions de fonctionnement (température ambiante, tension, humidité, assemblage, technologie du boitier, etc.).

Le premier objectif est de déterminer si l'on peut distinguer le vieillissement de deux processeurs à partir d'informations haut niveau telles que le nombre de cycles d'exécution ou le rapport du temps d'exécution sur le temps au repos. Cette information pourra être intégrée au sein de l'architecture multiprocesseur afin, par exemple, d'homogénéiser le vieillissement des processeurs.

Position du problème

Le deuxième objectif est d'observer les erreurs intermittentes qui peuvent apparaître avec l'activité des processeurs et le vieillissement. En effet, avant que des erreurs permanentes n'apparaissent à l'intérieur du processeur, des erreurs intermittentes peuvent survenir. Certains articles mentionnent un comportement de type *burst* [6, 7, 28, 29], c'est à dire que ces erreurs apparaissent en rafale pendant des durées allant de la nanoseconde à la seconde. Cependant, à notre connaissance, aucune étude publiée ne décrit précisément le phénomène. L'observation des erreurs intermittentes sur un cas réel de système embarqué est précieuse pour la mise en place de méthodes de test en ligne de ces erreurs.

Ainsi, il est nécessaire, dans un premier temps, de prouver qu'il est possible d'observer des erreurs intermittentes avant que le système ne soit complètement défaillant. Et dans un deuxième temps, il sera nécessaire d'extraire suffisamment d'informations pour modéliser les erreurs intermittentes (fréquence d'apparition, durée d'apparition, etc.).

Cahier des charges

Afin de répondre aux objectifs fixés, nous proposons la mise en place d'une plateforme expérimentale ayant les caractéristiques suivantes :

Mettre en place des tests accélérés, afin de réduire les durées de test et provoquer l'apparition d'erreurs intermittentes. En effet, nous savons que ces erreurs sont fortement liées au vieillissement. Or, ce dernier peut être amplifié par des tests accélérés. Cela permettra d'observer des erreurs intermittentes sur des délais très courts (quelques mois) comparés au temps normal de vieillissement (quelques années).

Permettre le test de plusieurs composants, afin d'obtenir des résultats représentatifs. Les tests seront réalisés sur une même technologie mais sur différents composants en parallèle. L'objectif n'est pas d'établir des statistiques sur le taux probable de défaillance pour un composant donné. Ainsi des centaines de composants ne seront pas nécessaires. Cependant, chaque expérience devra être réalisée sur au moins deux composants et un composant témoin. Le composant témoin ne sera pas vieillit et ne devra pas présenter d'erreur. Les deux

composants restants seront vieillis. Deux composants sont utilisés pour assurer une bonne observation des phénomènes, même si l'un d'eux se dégrade trop rapidement.

Générer une activité différente, pour deux lots de composants (voir premier objectif). Cela impose de pouvoir générer des activités différentes à l'intérieur des processeurs. L'activité des processeurs sera créée en exécutant des applications ayant des profils différents. Cette activité, sera également modulée en insérant des périodes de pause entre les applications.

Permettre l'observation d'erreurs intermittentes. Pour cela, la plateforme expérimentale doit permettre d'identifier et de différencier les différentes sources d'erreurs. C'est à dire que le nombre de sources d'erreurs doit être limité, afin de pouvoir concentrer les observations sur les systèmes testés.

Permettre le test de composants dans une technologie récente. En effet, dans le chapitre 1, nous avons vu que la probabilité d'observer des erreurs intermittentes augmente avec la diminution du facteur d'échelle. Les tests devront donc être réalisés dans la technologie la plus récente disponible au moment des expériences.

2.2 Tests accélérés des circuits intégrés

Cette expérimentation a pour but de faire apparaître des défauts de vieillissement, or ces défauts surviennent au bout de plusieurs années, dans des conditions normales de fonctionnement. Le but des tests accélérés est de modifier les conditions de fonctionnement afin d'accélérer l'apparition de ces défauts.

Cette partie, débute par un bref rappel sur les mécanismes de dégradation des circuits intégrés et leurs rapports avec la température. Nous verrons ensuite le principe des tests accélérés utilisés dans l'industrie. Cela permettra de définir le type de test accéléré à utiliser dans notre plateforme expérimentale. Au terme de cette partie, nous montrerons que la température est un vecteur de stress nécessaire et suffisant pour notre étude. Cependant, pour une caractérisation plus précise des fautes, ce seul stress ne serait pas suffisant.

Accélération du vieillissement

Le paragraphe 1.2.3 (page 20) décrit les principaux mécanismes de défaillance en jeu dans le vieillissement des circuits électroniques actuels. On peut citer les phénomènes d'*Électromigration* et de *Metal Stress Voiding* qui sont les principaux acteurs de la dégradation des lignes de métal, ainsi que les phénomènes de *Time Dependent Dielectric Breakdown*, de *Hot Carrier Injection* et de *Negative Bias Temperature Instability*, qui eux dégradent l'oxyde de grille des transistors.

Tout ces phénomènes possèdent une grande dépendance à la température ou à la tension (voir tableau 1.2 page 26). Ainsi l'augmentation de la température et/ou de la tension permet d'accélérer l'apparition des défauts de vieillissement. Cette caractéristique est utilisée par les méthodes de tests accélérés, pour faire vieillir artificiellement des lots de circuits intégrés dans l'industrie.

Tests accélérés industriels

Dans l'industrie les tests accélérés sont utilisés pour qualifier des lots et écarter les composants défectueux et fragiles. Parmi les tests les plus utilisés on trouve : Temperature Humidity (TH), Autoclave test, Temperature Cycling (TC), High Accelerated Temperature and Humidity Stress Test (HAST), High-Temperature Storage Life (HTSL), et Hot Temperature Operating Life (HTOL) [24, 27, 30–34]. Chacun de ces tests vise un ou plusieurs défauts spécifiques et correspond à un protocole particulier. Par exemple, le test HAST vise les défauts de corrosion en appliquant une température de 130°C et un taux d'humidité de 85% pendant 96 heures [27].

Les tests sont effectués sur une grande population de composants et permettent d'obtenir un taux de défaillance [1] pour le lot testé, la technologie et le type de test appliqué (ou le mécanisme de défaillance). Ce taux de défaillance s'exprime en FIT [2] et est calculé à partir de modèles statistiques. L'objectif est d'extrapoler le temps de défaillance, obtenu avec les tests accélérés, à celui qui correspondrait à des conditions normales de fonctionnement.

Ces données permettent aux fabricants, par exemple, de donner à leurs clients des données indicatives de durées de vie dans différentes conditions. Par exemple, Xilinx publie un document fiabilité [35] répertoriant les données de ces différents tests pour ses différents FPGA et ses différentes technologies. On y apprend par exemple, que pour les FPGA de type XC5VxXxxx, le résultat du test HTOL [3] indique un taux d'erreur de 6 FIT à température ambiante. Cela représente six composants défaillants parmi une population d'un million et après une durée de fonctionnement de mille heures.

Définition des conditions de tests accélérés

Les tests précédents sont réalisés sur des lots de composants importants et permettent d'obtenir des statistiques précises pour des mécanismes de défaillance spécifique. Dans notre cas, le but n'est pas de qualifier des composants spécifiques ou d'établir des durées de vie. Le but est de faire vieillir suffisamment les composants afin d'activer des défauts de vieillissement et ainsi pouvoir provoquer des erreurs intermittentes. Ainsi, il n'est pas nécessaire, dans notre cas, de se référer à un protocole de vieillissement industriel en particulier. Ainsi, nous pouvons réaliser notre étude sur des lots de tailles plus faibles mais sur une durée plus importante.

On sait que la température est un bon vecteur de vieillissement et que les principaux mécanismes de défaillance affectant les composants actuels sont l'*Électromigration* (EM), le *Time Dependent Dielectric Breakdown* (TDDB), le *Hot Carrier Injection* (HCI) et le *Negative Bias Temperature Instability* (NBTI). Ainsi, en se référant aux standards définis par le JEDEC [4] [27], il est possible de connaitre les conditions de stress relatifs à ces mécanismes de défaillance.

1. Failure rate en anglais
2. Failure In Time : nombre de défaillances par 10^9 heures
3. Test réalisé sur 548 composants, à une température égale à 125°C, en fonctionnement, pour une tension d'alimentation V_{DD} maximale et sur une durée équivalente de 2 092 148 heures.)
4. Joint Electron Devices Engineering Council (JEDEC) est une organisation développant des standards pour les semi-conducteurs

Le tableau 2.1 présente les conditions de stress en température et en tension pour les mécanismes de défaillance précédents. Mis à part le HCI, ces mécanismes sont accélérés avec une température supérieure à 150°C. Certains nécessitent une modification des tensions. Cependant, comme nous le verrons dans la suite de ce chapitre, nous ne pouvons pas modifier indépendamment V_{GS} et V_{DDmax}.

Le protocole de test accéléré utilisé dans nos expériences sera un stress constant en température. Pour éviter une défaillance trop rapide de nos composants, la température sera progressivement augmentée de 150°C à 200°C .

Mécanisme de défaillance	Température (°C)	Tension
EM	150-250	nominale
TDDB	25-200	$V_{GS} > V_{DDmax}$
NBTI	>125	$V_{GS} > V_{DDmax}$
HCI	25-30	augmentation de V_{DS}

TABLE 2.1 : *Conditions de stress standards pour la mise en évidence des phénomènes d'Électromigration (EM), de Time Dependent Dielectric Breakdown (TDDB), de Hot Carrier Injection (HCI) et de Negative Bias Temperature Instability (NBTI). [27]*

2.3 Description de la plateforme expérimentale

À la vue des objectifs présentés précédemment notre plateforme expérimentale doit pouvoir, premièrement étudier des processeurs, deuxièmement générer des activités différentes au sein des processeurs, et troisièmement générer un stress en température. Ainsi, la plateforme expérimentale sera décomposée en trois parties :

- les *circuits sous test* (CUT [5]), qui représentent des systèmes constitués d'un processeur, d'une mémoire et de périphériques de communication ;
- la *gestion du stress interne* (ISM [6]), qui permet de charger et d'exécuter différentes applications dans les CUT et de collecter les erreurs ;
- la *gestion du stress externe* (ESM [7]), qui permet d'accélérer le vieillissement des CUT en leur appliquant une température importante.

La figure 2.1 présente un schéma fonctionnel de la plateforme expérimentale. Chaque partie sera détaillée dans les paragraphes suivants.

5. Circuit Under Test
6. Internal Stress Manager
7. External Stress Manager

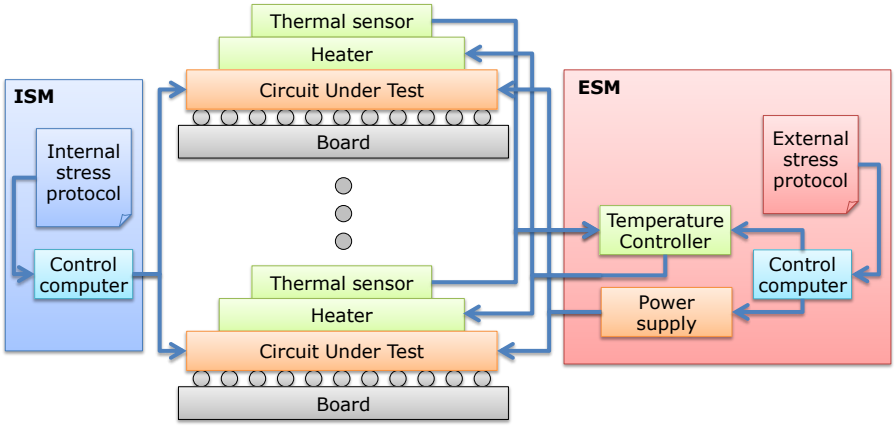


FIGURE 2.1 : *Schéma fonctionnel de la plateforme expérimentale.*

2.3.1 Description des circuits sous test

Le circuit à tester (CUT) comporte un cœur de processeur en technologie CMOS 65 nm. Chaque processeur est relié aux éléments suivants : une horloge, une alimentation, un port JTAG [8], un périphérique de communication et une mémoire. La figure 2.2 présente la configuration des cartes de test.

Le port JTAG permet la connexion entre l'ordinateur de contrôle et le processeur, et sera utilisé pour charger des applications dans le CUT.

La mémoire externe est utilisée par les applications et contient à la fois les instructions et les données. Nous verrons dans la partie suivante, que la mémoire n'est pas soumise à un stress en température.

Le périphérique de communication permet le transfert de données entre le CUT et l'ordinateur de contrôle, via un protocole Ethernet 100 Mbps.

La plateforme expérimentale est composée de six cartes AESV5FXT Avnet [36]. Chaque carte contient un FPGA Virtex5-FX et chaque FPGA contient un cœur de processeur PowerPC 440 [37] matériel en technologie 65 nm. Le processeur n'est pas programmé à partir des cellules FPGA, il est intégré comme une IP [9] dédiée.

2.3.2 Gestion du stress externe

Le rôle du *gestionnaire de stress externe* (ESM) est de contrôler la tensions d'alimentation et la température appliquée aux CUT. Pour cela, un *protocole de stress externe* est établi ; il indique pour chaque instant de l'expérience les valeurs de tension et de température à appliquer. De plus l'ESM est chargé d'enregistrer, tout au long des expériences, le courant consommé par les cartes et la température mesurée. Cela permet, d'une part, de conserver une trace qui peut

8. Standard IEEE 1149.1 appelé Standard Test Access Port and Boundary-Scan Architecture
9. Intellectual Property

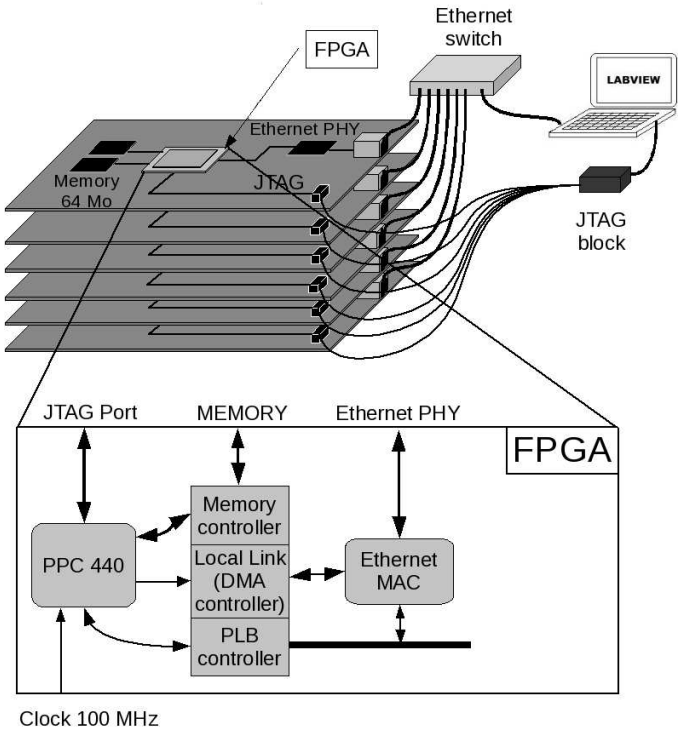


FIGURE 2.2 : *Schéma descriptif des circuits sous test (CUT).*

être exploitée a posteriori et, d'autre part, de surveiller que ces valeurs ne dépassent pas un seuil critique. En particulier, l'ESM veille à ce qu'il n'y ait pas d'emballement thermique, en régulant la température.

Gestion de la température

Afin d'augmenter la température de jonction des CUT, nous utilisons une résistance chauffante flexible Minco [38]. La résistance chauffante est collée sur le dessus du FPGA à l'aide d'un adhésif acrylique. La figure 2.3a illustre le montage utilisé. L'utilisation de résistances chauffantes permet d'appliquer le stress en température uniquement sur les CUT. Ainsi, les autres composants présents sur la carte ne sont pas stressés. De cette manière nous évitons que des erreurs venant des autres composants ne perturbent nos observations. Pour atteindre la température nécessaire et pour prévenir des variations de la température ambiante, les cartes sont placées dans une étuve régulée à la température de 50°C. Dans ces conditions, la température des CUT peut atteindre 200°C. Il est à remarquer que la résistance thermique des circuits intégrés récents étant très faible, la température de jonction des CUT est équivalente à la température de chauffage.

Un régulateur PID [10] contrôle le courant de chauffage, comme le montre la figure 2.3b. L'ordinateur de contrôle communique la température de consigne au régulateur par le biais d'une liaison série RS232.

10. Proportionnel Intégral Dérivé

Des générateurs de tension programmables commandent l'alimentation des cartes et des résistances chauffantes. L'ordinateur de contrôle est relié aux alimentations programmables par un bus GPIB [11]. Ainsi, l'alimentation des cartes peut être stoppée en cas de besoin, par exemple, si une des cartes dépasse une consommation maximale.

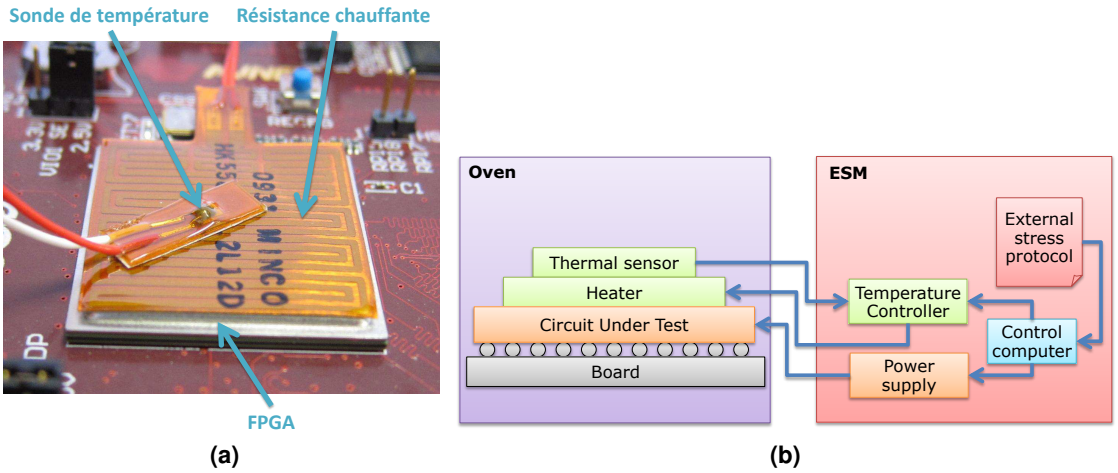


FIGURE 2.3 : *Description de la gestion du stress externe : (a) Implémentation matérielle du chauffage des CUT, avec un FPGA Xilinx Virtex5FX [39], une résistance chauffante Minco [38] et une sonde de température de type PT100. (b) Diagramme fonctionnel de l'ESM.*

2.3.3 Gestion du stress interne

L'ISM contrôle les applications exécutées par les CUT. Nous utilisons un ensemble d'applications pour créer de l'activité sur les différentes parties du processeur, et également pour détecter les erreurs dans ces régions. Le niveau d'activité dépend du rapport entre le temps passé à l'état actif et le temps passé à l'état de repos. La figure 2.4 montre deux niveaux de stress interne qui impliquent deux niveaux d'activité différents.

De la même manière que pour l'ESM, un *protocole de stress interne* est établi. Il indique pour chaque instant de l'expérience les applications à exécuter et le nombre d'exécutions de chaque application à effectuer. Ainsi, chaque application est exécutée à plusieurs reprises, comme le montre la figure 2.4. Les détails du protocole de test et des applications exécutées seront donnés dans la partie 2.4.

L'ISM s'exécute sur l'ordinateur de contrôle avec l'outil LabView [40]. À la fin de l'exécution de chaque application, il vérifie les résultats produits par chaque CUT afin de détecter les erreurs. Les étapes suivantes expliquent comment les applications sont chargées dans un CUT et comment les résultats sont vérifiés :

1. La première étape configure le FPGA en lui transmettant son fichier de configuration (bitstream). Il contient la configuration du processeur et des différents périphériques qui l'en-

11. Standard IEEE-488

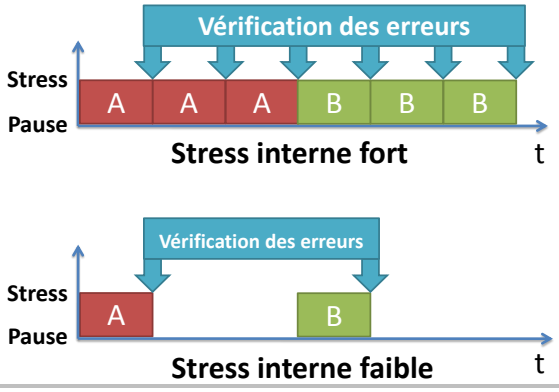


FIGURE 2.4 : *Exemples de stress interne. Les blocs A et B représentent une exécution de deux applications différentes. Les erreurs sont vérifiées à la fin de chaque application.*

tourent. Cette configuration est commune à toutes les applications. Ainsi, cette étape est effectuée une seule fois au début de l'expérience.

2. Ensuite, l'application est chargée dans la mémoire externe, puis le processeur est démarré à l'aide de commandes spécifiques envoyées par la liaison JTAG. Cette étape est effectuée à chaque changement d'application (sur la figure 2.4 au remplacement de l'application *(A)* par l'application *(B)*).
3. La communication Ethernet entre chaque CUT et l'ordinateur de contrôle est vérifiée. S'il n'y a pas de connexion, le processeur est remis à zéro et l'étape 2 est répétée. S'il n'y a pas de connexion la deuxième fois, le processeur est considéré comme défaillant. Dans ce cas, l'expérience est stoppée, les alimentations sont coupées et une demande de maintenance est envoyée.
4. L'ISM envoi une commande de démarrage au processeur. Celui-ci exécute une occurrence de l'application (une occurrence de *(A)* dans la figure 2.4). De cette façon, l'ISM synchronise l'exécution des applications sur les différents CUT.
5. Le processeur enregistre les résultats d'exécution dans la mémoire externe et calcule une signature (un mot). Une signature est calculée à partir des données produites. Cela, afin de limiter les communications et les erreurs de transmission.
6. L'ISM se met en attente des résultats des CUT. Celui-ci les envoie à trois reprises pour éviter des erreurs de communication transitoires. Si l'ISM ne reçoit aucune donnée après un temps limite, nous considérons que le processeur est en erreur. L'application est rechargée, retour à l'étape 2.
7. Le processeur retourne à l'étape 4 et attend la commande de démarrage avant d'exécuter de nouveau l'application.
8. L'ISM compare les résultats avec des valeurs de référence. Si une différence apparaît, l'ISM l'enregistre pour un post-traitement. La taille des données et le contenu sont ensuite vérifiés. Si aucun résultat n'est transmis, l'expérience est arrêtée et une demande de maintenance est envoyée.
9. En fonction du *protocole de stress interne* trois choix sont alors possible. Premièrement, l'application *(A)* doit de nouveau s'exécuter, dans ce cas, l'ISM envoie la commande de dé-

marrage à chaque processeur. Les processeurs exécutent la même application *(A)* et passent à l'étape 5. Deuxièmement, l'application *(A)* doit être arrêtée, dans ce cas, la prochaine application *(B)* est chargée dans la mémoire (étape 2). Troisièmement, l'activité des processeurs doit être modulée (stress interne faible), dans ce cas, ils sont arrêtés en attendant de changer d'application (voir figure 2.4).

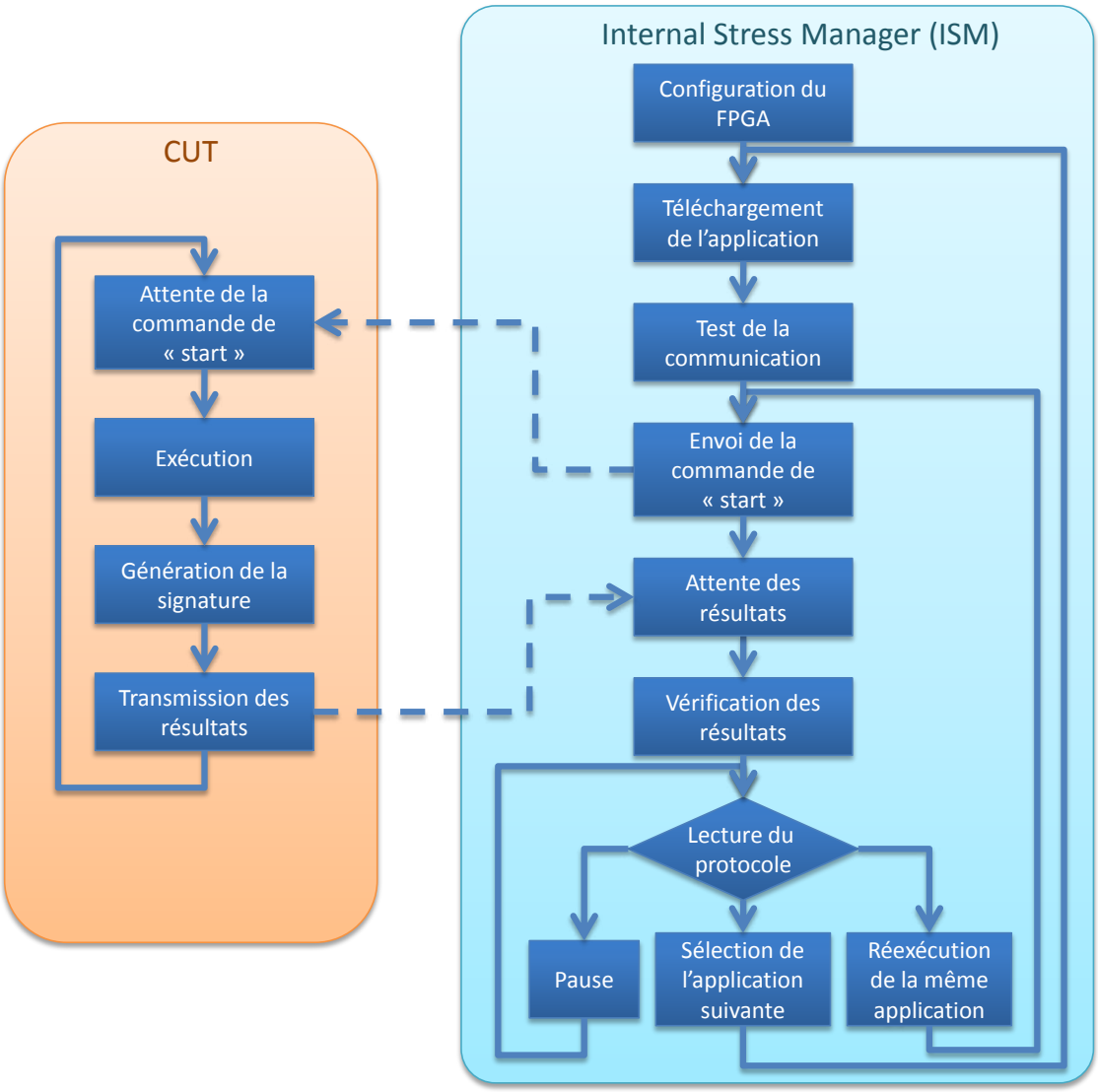


FIGURE 2.5 : *Étapes de la gestion du stress interne.*

2.3.4 Gestion des erreurs

Sur la plateforme expérimentale, des erreurs peuvent survenir à n'importe quel stade de l'expérience. Nous énumérons quatre types d'erreurs :

Erreur de chargement de la configuration du FPGA : cette erreur peut se produire lorsque le fichier de configuration (bitstream) est chargé dans le FPGA. Si le FPGA ne peut plus être configuré alors le CUT est considéré comme défaillant, il est retiré de l'expérience. En effet, plusieurs cartes étant sur la même chaîne JTAG, nous avons observé que le composant défaillant perturbe les composants sains. Dans ce cas de fausses erreurs peuvent être observées sur les composants sains.

Erreur de chargement des applications : cette erreur peut se produire lorsque le programme est chargé dans la mémoire des CUT. C'est le programme chargé de transférer les applications via JTAG qui émet cette erreur. Ce type d'erreur est principalement apparu après la défaillance d'un CUT.

Erreur de communication : ce type d'erreur se produit lorsque le programme Labview tente de communiquer avec les CUT pour obtenir les résultats d'exécution. La communication se fait via un port Ethernet avec le protocole TCP/IP. Cette erreur peut être causée par une erreur dans le processeur, dans le contrôleur DMA, dans le contrôleur Ethernet ou le contrôleur mémoire. Par conséquent, nous ne pouvons pas déterminer précisément où l'erreur se produit. Toutefois, cette erreur peut être causée par un défaut intermittent.

Erreur de calcul : cette erreur se produit si une faute intermittente apparaît lorsque le processeur exécute des calculs. Dans ce cas, les résultats produits diffèrent de ceux attendus, une erreur intermittente est détectée.

Les erreurs de calcul sont les erreurs les plus couramment observées dans nos expériences et les plus représentatives des erreurs intermittentes. Nous allons, par la suite, nous concentrer uniquement sur ce type d'erreur.

2.4 Protocole expérimental

Deux types de stress sont considérés dans la plateforme expérimentale : le stress interne et le stress externe. Un de nos objectifs est de déterminer si l'activité des processeurs a un impact sur l'apparition des erreurs intermittentes. Pour cela, deux groupes de trois cartes subissent un stress interne différent. Dans chaque lot, deux cartes sont soumises au même stress externe (voir figure 2.6), et une carte ne subit aucun stress externe destructif (carte témoin).

Le stress externe correspond à une température constante comprise entre 125°C et 190°C. Cependant les résultats effectués à une température inférieure à 160°C ne seront pas pris en compte par la suite car aucune erreur de calcul n'a été détectée à ces températures.

La partie suivante décrit les applications utilisées dans le protocole de stress interne.

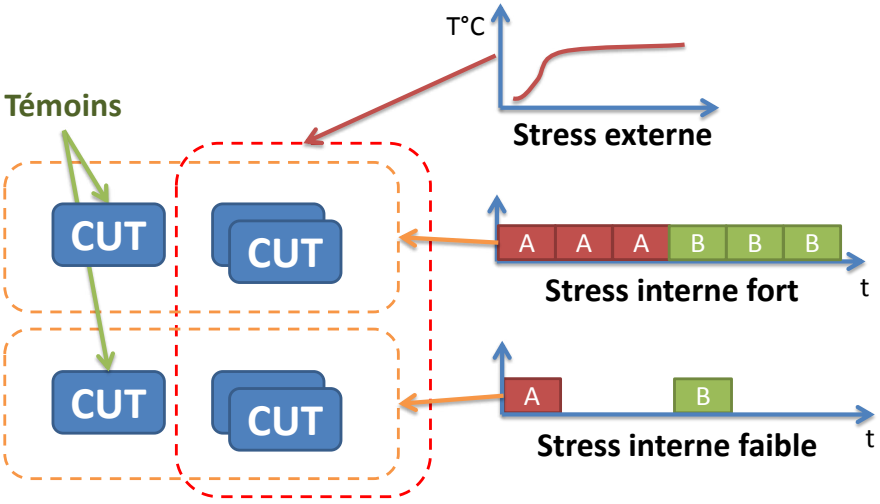


FIGURE 2.6 : *Protocole de stress. Deux groupes de trois cartes subissent un stress interne différent. Dans chaque groupe deux cartes sont soumises au même stress externe, et une carte ne subit aucun stress externe destructif (carte témoin).*

Description des applications

Le protocole de stress interne est composé de sept applications. *QuickSort* est une application de la suite MiBench [41]. *DCT*, *Quant* et *FIR* sont des applications embarquées classiques utilisées dans le traitement du signal. *TestFunct*, *TestFunct2*, *TestFunct3* sont des applications implémentant différentes méthodes de Sofware-Based Self-Test (SBST) [42].

Ces trois applications ont pour but de tester les unités fonctionnelles du processeur à partir de vecteurs de test. Dans le cas de *TestFunct* et *TestFunct2*, les vecteurs ont été calculés aléatoirement hors-ligne et sont stockés en mémoire. Dans le cas, de *TestFunct3* les vecteurs sont générés aléatoirement en ligne. Les vecteurs sont ensuite "envoyés" sur les entrées des unités fonctionnelles à l'aide des instructions de l'ISA [12] (*add*, *sub*, *or*, etc.).

La différence entre ces trois applications se situe au niveau de la communication avec la mémoire. Dans le cas de *TestFunct* et *TestFunct2*, pour chaque test effectué deux transferts mémoire sont nécessaires (un pour charger le vecteur et un pour enregistrer le résultat). Dans le cas de *TestFunct3*, tous les calculs sont effectués à partir des registres internes, la communication avec la mémoire est donc très faible et l'activité du processeur est maximale.

12. Instruction Set Architecture

La différence entre *TestFunct* et *TestFunct2* vient de la programmation de l'algorithme. *TestFunct2* est optimisé afin de produire une activité supérieure à *TestFunct*. Cependant, le nombre de transferts mémoire est identique pour les deux applications.

Le fonctionnement des méthodes de SBST sera détaillé plus précisément dans le chapitre 3.

Le tableau 2.2 montre les caractéristiques des ces applications. En particulier, le tableau montre les différences dans le nombre de Load/Store par seconde et le nombre d'instructions par seconde pour chaque application. Ces deux valeurs permettent de caractériser le profil de chaque application. Par la suite, nous corrélerons le taux d'erreur observé avec ces informations. Cela nous permettra d'analyser quel paramètre a le plus d'impact sur le taux d'erreur observé dans nos expériences.

Ces applications ont des comportements très différents. Par exemple, seulement $0,0012\%$ des instructions effectuées par l'application *TestFunct3* sont des Load/Store, contre $34,1\%$ pour *Qsort*. Cela confirme que l'application *TestFunct3* utilise essentiellement les registres internes et communique peu avec la mémoire externe, à la différence de l'application *Qsort*. Ainsi, l'ensemble des applications est utilisé pour couvrir une diversité de profil d'applications.

Dans le cas d'un stress interne fort, chacune des sept applications s'exécute en continu pendant une heure, ce temps est nommé CET (Continuous Exeution Time). Par exemple, l'application *FunctionnalTest3* s'exécute 53 fois pendant une heure. Le temps nécessaire pour exécuter toutes les applications est égal à sept heures. C'est la période PET (Periodic Execution Time). Les applications sont exécutées en continu jusqu'à ce que tous les CUT soient défaillants. Ainsi, chaque application est exécutée périodiquement pendant une heure (CET) toutes les sept heures (PET). La figure 2.7 montre ce protocole de stress interne.

N°	Applications	# load/store par s	# Instr. par s	Durée (s)
A	Qsort	8,567,619.96	25,157,627.08	9.28
B	FIR	5,113,756.02	12,600,864.70	7.60
C	Quant	3,982,256.49	10,202,307.42	11.48
D	TestFunct2	1,323,048.81	6,084,324.06	1.52
E	TestFunct	925,207.17	3,544,534.22	2.18
F	DCT	27,544.99	180,340.58	1.98
G	TestFunct3	110.17	9,531,901.79	67.84

TABLE 2.2 : *Caractéristiques des applications.*

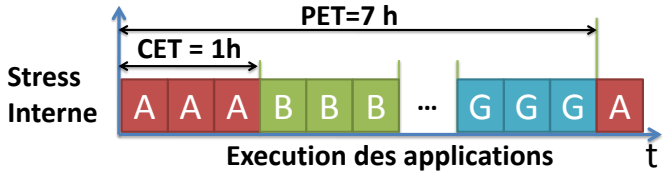


FIGURE 2.7 : *Protocole de stress interne fort. Chacune des sept applications s'exécute en continue pendant une heure (CET) toutes les sept heures (PET). CET=Continuous Execution Time and PET=Periodic Execution Time. Dans le cas d'un test interne faible, les applications ne sont exécutées qu'une seule fois pour chaque CET.*

2.5 Résultats

Cette partie présente et analyse les résultats de la plateforme expérimentale. Plus précisément, nous débuterons en présentant les conditions expérimentales de l'expérience. C'est à dire, que nous détaillerons l'historique expérimental des cartes ainsi que la manière dont elles ont été déclarées défaillantes.

Par la suite, nous analyserons les erreurs apparues lors de l'expérience, et nous les corrélerons avec les différents stress appliqués. Premièrement, nous verrons comment les erreurs intermittentes évoluent avec l'activité des processeurs. Deuxièmement, nous comparerons les erreurs observées avec les différents applications.

Pour finir, nous étudierons dans le détail comment apparaissent les erreurs intermittentes, ce qui les caractérise, et comment elles évoluent en fonction du temps.

2.5.1 Conditions expérimentales

Historique

Avant de présenter les résultats de l'expérience présentée dans la partie précédente, il est nécessaire de présenter l'historique expérimental des cartes. En effet, un première expérience à précédée celle que nous présentons dans cette partie. Cette expérience a été réalisée à 145°C pendant six mois et n'a présenté aucune erreur de calcul sur les différents CUT. Par conséquent, nous pouvons considérer que les erreurs analysées par la suite ne sont pas dues, ni à une mauvaise configuration du FPGA, ni à des problèmes logiciels. Cela confirme que les erreurs de calcul observés résultent directement de l'augmentation de la température et du vieillissement induit par celle-ci. Toutefois, au cours de cette période, les différents CUT ont été chauffés et donc soumis à un vieillissement accéléré.

L'expérience qui nous intéresse a été réalisée en appliquant aux CUT un stress interne de 170°C. De plus, nous pouvons remarquer que les CUT témoins n'ont montrés aucune erreur.

Au lancement de l'expérience les CUT ne sont pas alimentés. La température est progressivement augmentée jusqu'à l'obtention de la température de consigne souhaitée. Cette durée est de 25 minutes. Une fois la température atteinte, les cartes sont alimentées et l'expérience peut commencer.

Les paragraphes suivants montrent comment les cartes ont été considérées comme défaillantes et après quelle durée.

Défaillance des CUT

Le tableau 2.3 indique le temps avant défaillance en fonction des différents CUT. Nous pouvons remarquer que les CUT 1 et 6 n'ont pas de temps de défaillance. En effet, ce sont les CUT témoins et ils n'ont présenté aucune erreur.

Les CUT 2 et 3 ont été soumis à un stress interne fort (lot 1), alors que les CUT 4 et 5 ont été soumis à un stress interne faible (lot 2).

Pour un même lot nous constatons que les durées avant défaillance peuvent être très différentes. Dans le lot 1, le CUT 3 est devenu défaillant après une durée de 80h alors que le CUT 4 est devenu défaillant après une durée de 241h. Cette dernière durée sera la base de temps utilisée pour présenter les résultats, les erreurs ayant principalement été observées dans le lot 1.

La fin de l'expérience a été donnée avec la défaillance du CUT 4 après 308h.

Lot	CUT	Stress interne	Stress externe	Temps avant défaillance
1	1	fort	NON (témoin)	NA
1	2	fort	OUI	241h
1	3	fort	OUI	80h
2	4	faible	OUI	308h
2	5	faible	OUI	173h45
2	6	faible	NON (témoin)	NA

TABLE 2.3 : *Temps avant défaillance en fonction des cartes*

Comportement des défaillances

Un CUT est considérée comme défaillant quand le processeur ne peut plus être programmé. Dans ce cas la carte est retirée de la plateforme.

Le comportement des CUT après défaillance est similaire. Nous avons observé que la défaillance n'est pas totalement destructive mais dépend de la température. En effet, les CUT défaillants retrouvent leur fonctionnalité à température ambiante. Cependant, en augmentant progressivement la température, ils redeviennent défaillants au-delà de 125°C.

D'une part, la température accélère les mécanismes de défaillance qui ont un impact sur les délais de propagation du composant. Et d'autre part, la température a un lien direct avec les délais de propagation dans les lignes. Ce résultat semble donc cohérent, cependant on peut s'interroger sur l'amplitude de cette défaillance. En effet, les CUT sont passés brutalement d'une température limite de fonctionnement supérieure à 170°C à une température limite inférieure à 125°C. La dégénérescence du circuit ne semble donc pas être linéaire.

2.5.2 Taux d'erreurs intermittentes

Le premier résultat présente l'ensemble des erreurs détectées au cours de l'expérience, tout CUT confondus. Ainsi, la figure 2.8 indique le nombre total d'erreurs intermittentes de calcul observées. Chaque point représente le nombre total d'erreurs observé pendant une période de sept heures (PET).

Cela montre qu'il est possible d'observer des erreurs intermittentes à un niveau système et très tôt avant que le système ne tombe en panne. Nous pouvons constater que des erreurs ont été

observées dès la première période PET. Durant cette période, 148 erreurs ont été observées toutes applications confondues. Nous verrons par la suite comment les erreurs se répartissent suivant les applications.

Plus précisément, la première erreur de calcul a été détectée après $3,12$ heures sur le CUT 2.

La ligne pointillée sur la figure 2.8 montre la moyenne du nombre d'erreurs détectées au cours de chaque période de PET. Nous pouvons voir que le nombre d'erreurs intermittentes augmente avant la défaillance générale de tous les CUT. Entre 100h et 250h d'expérience, la moyenne du nombre d'erreurs intermittentes augmente linéairement d'un coefficient 3.67 erreurs par heure. Le creux observé entre 50 et 100 heures correspond à la date de défaillance d'un des CUT soumis à un fort stress interne (CUT 3, voir tableau 2.3). C'est pour cela que, durant cette période, moins d'erreurs sont observées.

Conclusion : Ces résultats confirment l'existence d'erreurs intermittentes et justifient l'utilisation de méthodes de test en ligne pour détecter ces erreurs. En effet, si aucune action n'est prise, les erreurs semblent augmenter progressivement jusqu'à la défaillance des CUT.

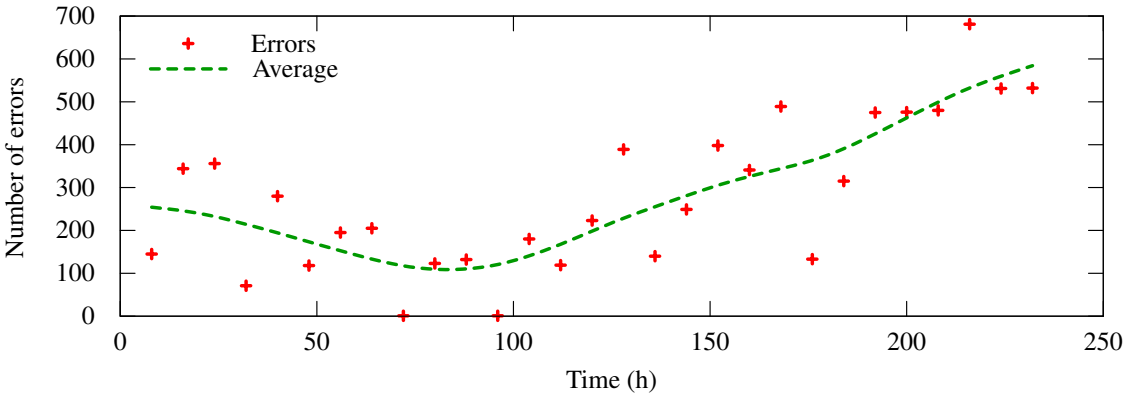


FIGURE 2.8 : *Courbe mettant en évidence les problèmes intermittents et représentant le nombre d'erreurs intermittentes de calcul observées sur tous les CUT à 170 °C. Chaque point représente le nombre d'erreurs observé pour un PET. C'est à dire pour une période de sept heures.*

2.5.3 Impact de l'activité des processeurs sur les erreurs intermittentes

Un de nos objectifs est d'analyser l'impact de l'activité des processeurs sur l'apparition des erreurs. En effet, nous avons vu dans le chapitre 1 que le lien entre activité et vieillissement n'est pas trivial et peut varier d'une technologie à une autre. De plus, les observations faites dans cette partie ne s'appliquent qu'à la technologie dans laquelle les CUT sont réalisés (Xilinx 65 nm).

Pour répondre à cet objectif, ce paragraphe compare le nombre d'erreurs observées sur les deux lots. Les CUT soumis à un faible stress interne exécutent chaque application une fois toutes les heures. Afin de comparer le nombre d'erreurs intermittentes observées en fonction du stress

interne, seules les erreurs observées lors de la première exécution de chaque programme sont conservées.

Ainsi, la figure 2.9 montre le nombre d'erreurs de calcul détectées lors de la première exécution de chaque programme pour les CUT soumis à un fort stress interne et pour les CUT soumis à un faible stress interne. On observe un total de 25 erreurs pour les CUT soumis à un fort stress interne et un total de 2 erreurs pour les cartes à faible stress interne.

Le tableau 2.4 compare le taux moyen d'erreurs des lots 1 et 2 en prenant le rapport du nombre d'erreurs de calcul détectées lors de la première exécution de chaque programme sur le temps séparant le début des expérimentations et la défaillance de la dernière carte (241 heures). Et on observe un rapport de $12,45$ entre les deux CUT soumis à des activités différentes.

Conclusion : Les cartes soumises a un stress interne fort génèrent plus d'erreurs que les cartes soumises à un stress interne faible. L'activité des processeurs a bien un lien avec la génération d'erreurs intermittentes.

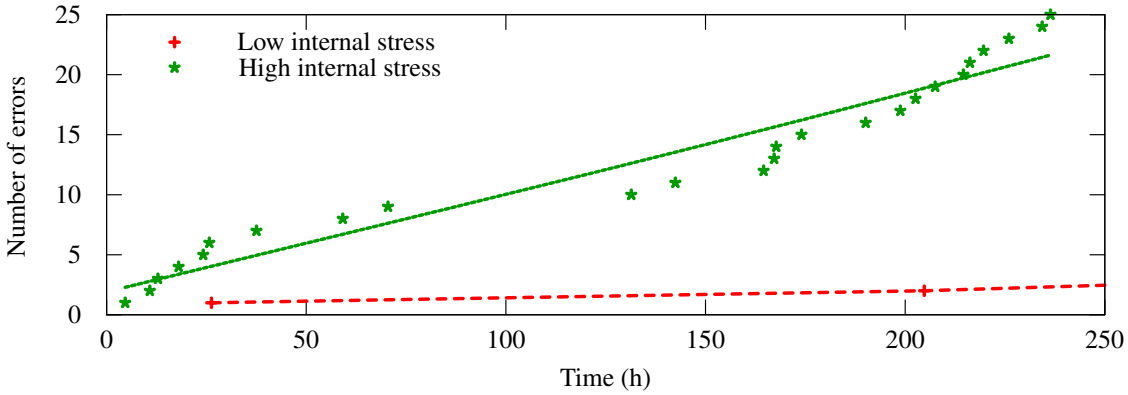


FIGURE 2.9 : *Nombre d'erreurs de calcul détectées lors de la première exécution de chaque programme (chaque heure) sur toute la durée des expériences.*

Lot	Stress interne	Taux moyen d'erreur de calcul par heure
1	Fort	0.08
2	Faible	0.006

TABLE 2.4 : *Taux moyen d'erreurs de calcul par heure et par lot. Pour chaque lot la moyenne a été réalisée sur tous les CUT confondus.*

2.5.4 Impact des erreurs intermittentes sur les applications

Ce paragraphe compare la sensibilité des différentes applications exécutées sur les CUT avec les erreurs intermittentes observées. Ainsi, les résultats combinent les erreurs observées sur tous les CUT des deux lots.

La courbe 2.10a représente une estimation par noyau [43] (kernel density estimation en anglais) de la densité de probabilité d'observer une erreur intermittente au cours du temps. Cette estimation est aussi appelée méthode de Parzen-Rozenblatt, c'est une méthode non-paramétrique qui permet d'estimer la densité de probabilité en tout point. Plus il y a d'observations au voisinage d'un point et plus la densité est élevée. Ainsi les pics représentent les instants de forte densité d'erreur. Dans notre cas, cela nous permet d'observer si la densité d'erreurs évolue en fonction du temps ou si elle est constante au cours du temps.

En particulier, nous constatons que tous les programmes n'ont pas de pics de densité aux mêmes instants. Ainsi, les programmes les plus affectés par les erreurs intermittentes évoluent en fonction du temps. Jusqu'à 100h d'expérience, la plus forte densité observée concerne les applications *FIR* et *TestFunct*, alors qu'à la fin de l'expérience cela concerne les applications *Qsort* et *TestFunct2*.

La courbe 2.10b présente un cumul des erreurs de calculs observées au cours du temps sur tous les CUT et permet de comparer le nombre d'erreurs observées sur les différents programmes.

On observe notamment que le programme *Qsort* est plus soumis à des erreurs intermittentes et que l'évolution du nombre d'erreurs observées au court du temps est constante. On peut également observer que le programme *TestFunct3* est le programme sur lequel le moins d'erreurs a été observé au cours des expériences.

Plus précisément, nous constatons que les applications avec le plus grand nombre d'erreurs sont aussi celles qui effectuent le plus grand nombre de Load/Store par seconde (voir tableau 2.2). Le nombre de Load/Store par seconde représente le trafic avec la mémoire externe. Ainsi, les applications effectuant le plus de trafic avec la mémoire externe semblent être plus sensibles aux erreurs intermittentes.

Conclusion : Tous les programmes semblent être affectés par les erreurs intermittentes. Cependant, le nombre d'erreurs observées ne semble pas être lié au nombre d'instructions par seconde, mais plutôt au nombre de load/stores par seconde.

2.5.5 Mise en évidence de burst d'erreurs intermittentes

Les résultats précédents ont analysé des nombres moyens d'erreurs en combinant des observations sur plusieurs CUT voire sur plusieurs lots. En effet, l'objectif était d'évaluer quantitativement l'impact des différents stress sur le nombre moyen d'erreurs observées. Dans cette partie nous analysons l'apparition des erreurs et tentons d'extraire les caractéristiques des erreurs observées. Ainsi, les observations seront faites sur un seul CUT et une seule application : le CUT 2 et l'application *TestFunct3*. Cependant, les résultats s'appliquent à toutes les applications.

Premièrement, observons comment les erreurs se manifestent au sein d'une même application. La figure 2.11 représente l'accumulation des erreurs observées sur le CUT 2 pour le programme *TestFunct3* tout au long de l'expérience.

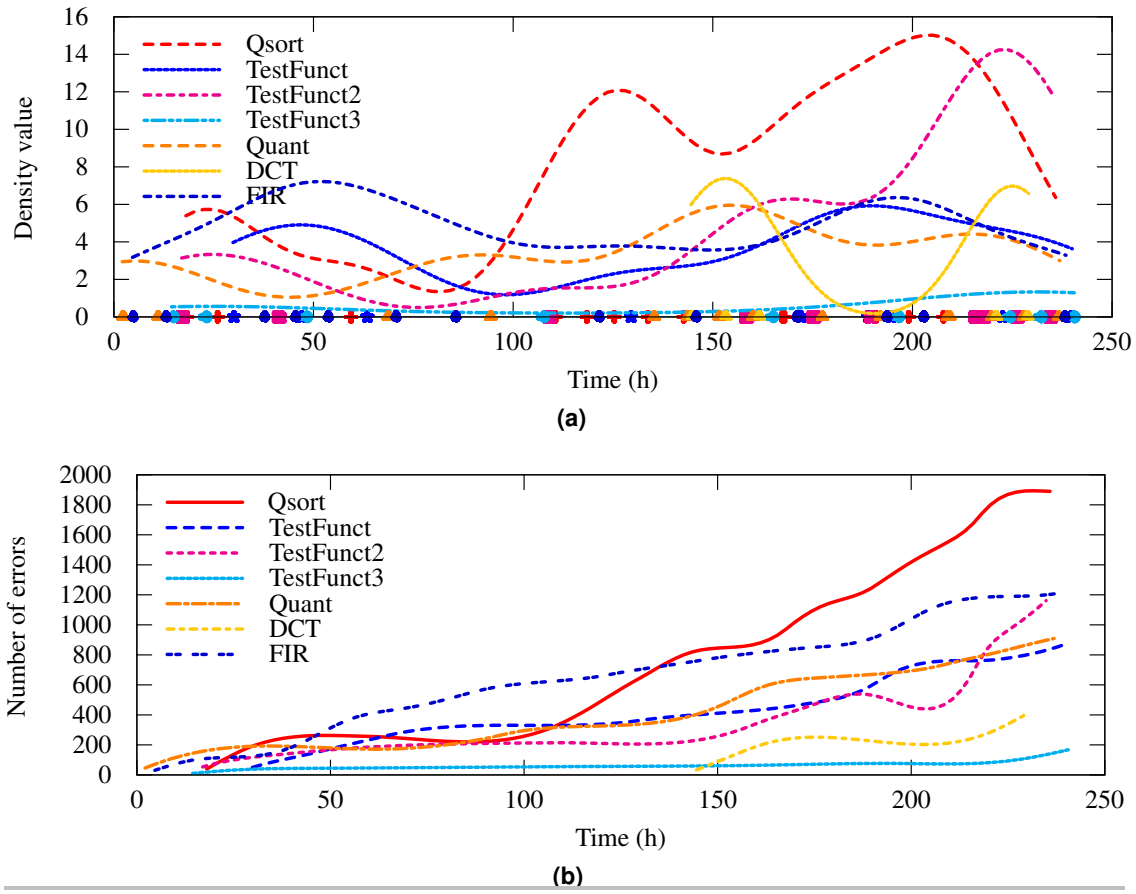


FIGURE 2.10 : *Courbes comparatives des liens entre erreurs intermittentes et programmes exécutés sur les cartes à fort stress interne.*
(a) Estimation par noyau de la densité de probabilité d'observer une erreur intermittente en fonction du temps pour chacune des applications.
(b) Cumul d'erreurs en fonction des applications.

Chaque programme s'exécute périodiquement une heure (CET) toutes les sept heures (PET), ainsi chaque programme s'est exécuté au maximum 34 fois au cours de l'expérience. La figure 2.11 montre uniquement 5 périodes de l'application avec des erreurs. Cela indique que l'application n'est pas erronée pour chacune de ses périodes d'exécution. En outre, une fois que l'erreur est déclenchée, si aucune mesure n'est prise pour y mettre fin, le programme continue de produire des erreurs intermittentes.

Le fait que plusieurs erreurs se produisent en rafale pendant l'exécution d'une même application et pendant une courte période est appelée *burst*. Nous nommerons ce phénomène par le terme anglais qui se prête mieux au phénomène et qui traduit mieux sa soudaineté. Selon nos observations, toutes les erreurs de calcul arrivent en *burst*.

Ici, l'erreur est corrigée par l'arrêt du processeur, lors du chargement du programme suivant (de l'application *(X)* à *(Y)*). Par conséquent, ce résultat confirme que des techniques de suspension [6] utilisées pour récupérer des erreurs intermittentes peuvent être utilisées. Ces techniques

consistent à suspendre le processeur pendant un certain temps afin de stopper les erreurs intermittentes.

Pour être encore plus précis, analysons un des burst d'erreurs intermittentes. La figure 2.12 montre une trace d'exécution pour le programme *TestFunct3* en présence d'erreurs intermittentes au cours d'un CET.

Chaque point correspond au résultat de chaque exécution (comme expliqué dans la figure 2.4). Le résultat est soit *erroné*, soit *correct*. Cette figure met en évidence le caractère répétitif des erreurs intermittentes.

Nous pouvons caractériser une erreur intermittente par deux paramètres : la durée moyenne séparant deux erreurs et la durée moyenne de maintient de l'erreur. Dans notre cas, le temps moyen avant erreur est d'environ $2,1$ exécutions et la durée moyenne des erreurs intermittentes est d'environ $1,5$ exécutions. Ainsi, pour chaque *burst* d'erreurs, il est possible de calculer ces valeurs. La partie suivante étudie l'évolution de ces caractéristiques au cours du temps.

Conclusion : Toutes les erreurs observées apparaissent sous la forme d'un *burst* d'erreurs et seul l'arrêt du processeur semble stopper l'apparition des erreurs. De plus, les erreurs intermittentes peuvent être caractérisés par deux paramètres : la durée moyenne séparant deux erreurs et la durée moyenne de maintient de l'erreur.

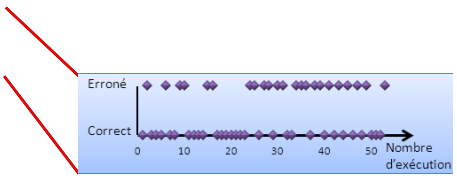


FIGURE 2.11 : *Nombre d'erreurs observées sur un CUT pour le programmeTestFunct3 tout au long de l'expérience. Chaque ensemble d'erreurs correspond à un burst d'erreurs intermittentes.*

2.5.6 Évolution des caractéristiques des bursts dans le temps

Le paragraphe précédent met en évidence des *bursts* d'erreurs intermittentes et montre, en particulier, une trace d'exécution d'un programme. A partir de cette trace, il est possible de caractériser l'erreur intermittente par deux paramètres, la durée moyenne séparant deux erreurs et la durée moyenne de maintient de l'erreur. La figure 2.13 illustre ces caractéristiques.

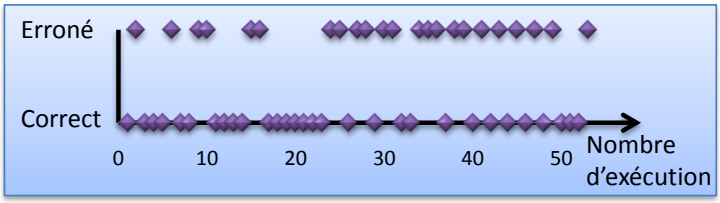


FIGURE 2.12 : *Trace d'exécution pour le programmeFunctionnalTest3 en présence d'un burst d'erreurs intermittentes, au cours d'un (CET) de une heure. Chaque point correspond au résultat de l'exécution : soit erroné, soit correct.*

Nous avons vu précédemment que toutes les erreurs observées sont apparues au cours d'un burst. Ainsi, pour chaque groupe d'erreurs observées sur une application, après un CET, il est possible de calculer ces deux paramètres. Afin d'analyser leur évolution au cours du temps, nous avons présenté sur une même courbe chacun de ces paramètres toutes applications confondues. La figure 2.14a montre l'évolution de la durée moyenne de maintient des erreurs intermittentes (TMEDE) au cours du temps, et la figure 2.14b présente l'évolution de la durée moyenne entre deux erreurs intermittentes.

La figure 2.14a montre que la durée moyenne de maintien des erreurs intermittentes (TMEE) est inférieure à 80 exécutions d'application en début d'expérience et supérieure à 120 à la fin. Cela représente une augmentation supérieure à 50% et cette augmentation est commune à toutes les applications. Cela montre, que la durée des erreurs augmente avec le vieillissement.

Concernant, l'évolution de la durée moyenne entre deux erreurs intermittentes, la figure 2.14b montre que cette durée est globalement constante sur la durée de l'expérience, mais qu'elle tend à diminuer sur la fin de l'expérience.

Conclusion : Nous montrons que la durée moyenne de maintien des erreurs augmente avec le vieillissement, mais que la durée moyenne entre deux erreurs intermittentes tend à rester constante. En particulier, cela nous aidera par la suite dans la modélisation des erreurs intermittentes.

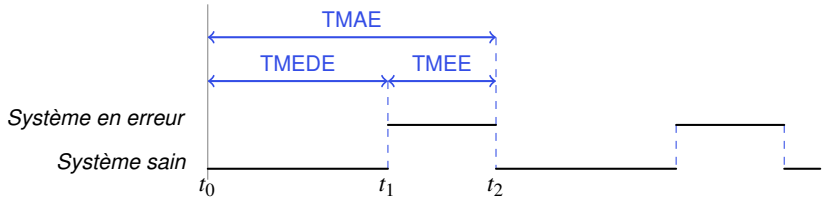


FIGURE 2.13 : *Les bursts d'erreurs intermittentes peuvent être caractérisés par les paramètres suivants : TMEDE = Temps moyen entre deux erreurs, TMEE = Temps moyen en erreur, TMAE = Temps moyen avant erreur.*

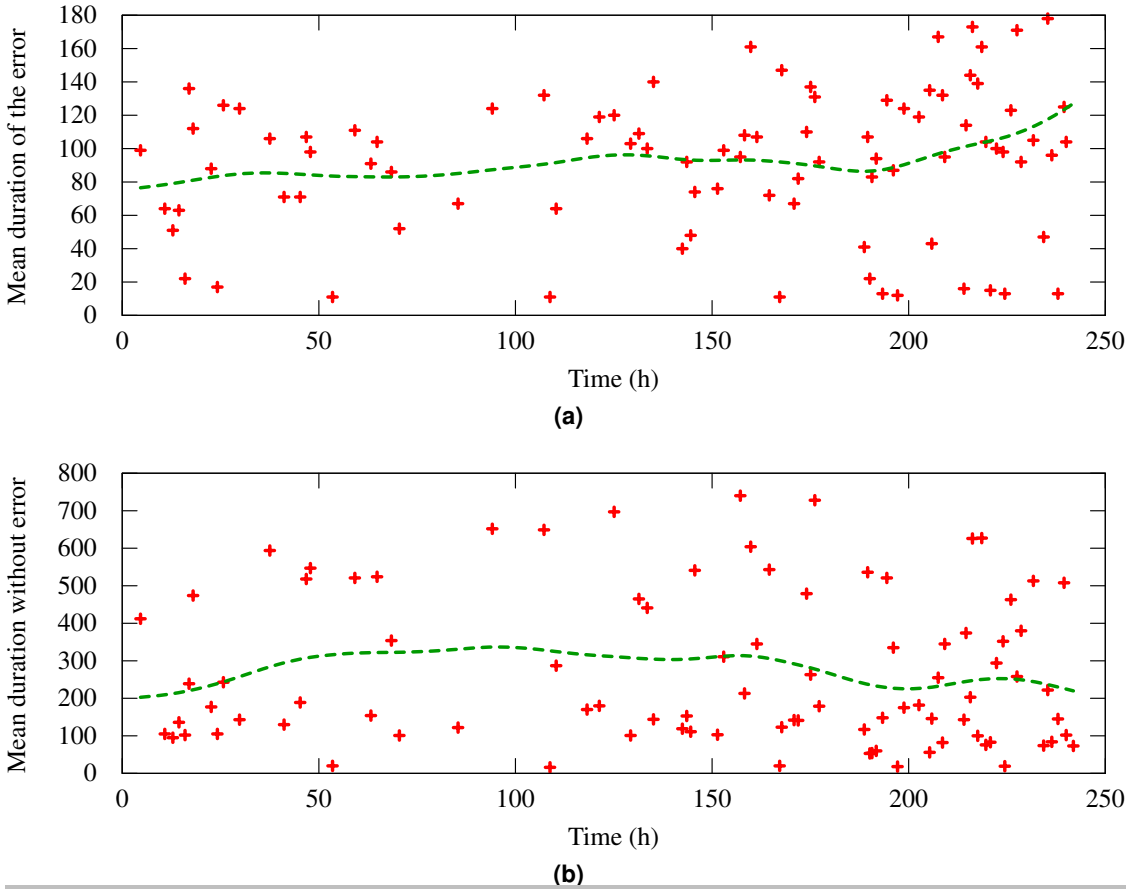


FIGURE 2.14 : *(a)Durée moyenne de maintien des erreurs intermittentes en nombre d'itération de chaque application (TMEE). (b) Durée moyenne entre deux erreurs intermittentes en nombre d'itération de chaque application (TMEDE).*

2.5.7 Implications pour la gestion du test en ligne

Comme nous l'avons mentionné dans l'introduction de ce mémoire, notre objectif est de développer des méthodes de test en ligne adaptées à la fois aux erreurs intermittentes et aux architectures multiprocesseurs. Dans ce sens, les résultats de ce chapitre permettent d'apporter plusieurs informations utiles pour la suite.

Premièrement, nous avons prouvé qu'il est possible d'observer des erreurs intermittentes avant l'apparition d'erreurs permanentes. Et cela suffisamment tôt pour envisager la restauration des processeurs. Cela confirme que l'utilisation d'un test en ligne doit être mise en place.

Deuxièmement, nous savons que les erreurs intermittentes apparaissent en *burst* et que seul l'arrêt du processeur permet de stopper ces erreurs. Tant que le processeur est en activité les erreurs sont maintenues. Cela peut être utilisé en plus du test en ligne comme méthode de restauration du système. Une fois l'erreur détectée, le processeur peut être arrêté le temps de stopper l'apparition de l'erreur.

Troisièmement, nous avons vu que les *burst* d'erreurs intermittentes peuvent être représentés par deux paramètres, leur durée moyenne de maintien et la durée moyenne entre deux erreurs. Basé sur ces paramètres, nous pourrons modéliser ces erreurs par un modèle de Markov afin d'évaluer la probabilité de détection de plusieurs méthodes de test en ligne.

Quatrièmement, nous avons observé plus d'erreurs sur les processeurs soumis à la plus forte activité. Dans une architecture multiprocesseurs, ce résultat permettra de faire l'hypothèse que les processeurs soumis à la plus forte activité ont la plus grande probabilité d'être en erreur.

2.6 Conclusion

Jusqu'à présent, aucune étude n'a observé des erreurs intermittentes sur un système embarqué dans une technologie actuelle. Ce chapitre a permis de définir une plateforme expérimentale capable d'observer des erreurs intermittentes. Pour cela, des processeurs en technologie 65 nm sont soumis, d'une part à une température supérieure à 160°C et d'autre part à l'exécution d'un ensemble d'applications. Ces deux paramètres représentent respectivement un stress externe et un stress interne. Le stress en température permet d'accélérer le vieillissement des processeurs et ainsi de générer des erreurs intermittentes. Le stress interne permet de sensibiliser les différentes parties du processeur et permet de générer plusieurs profils d'activité différents.

Nous avons pu confirmer que les erreurs intermittentes peuvent être observées avant l'apparition d'erreurs permanentes. Nos résultats confirment la présence de défauts intermittents très tôt avant la période d'usure du circuit. De plus, nous montrons que les circuits intégrés soumis à la plus forte activité présentent le plus grand nombre d'erreurs intermittentes. Cependant, toutes les applications utilisées n'ont pas présenté le même nombre d'erreur, ainsi l'observation des erreurs intermittentes dépend aussi des applications.

Concernant le mode d'apparition des erreurs intermittentes, toutes nos observations ont montré une apparition de type *burst*. Les erreurs apparaissent en rafale et seul l'arrêt des processeurs semble les stopper.

Ces informations seront utiles pour mettre en place un système de détection en ligne des erreurs intermittentes.

CHAPITRE 3

Méthodes de détection en ligne des erreurs

Sommaire

LES DEUX CHAPITRES précédents ont permis de situer le contexte de ce mémoire. Nous avons vu, dans le premier chapitre que plus la technologie évolue et plus il est difficile de fabriquer des circuits intégrés sans aucun défaut. Ces défauts ne sont pas forcément visibles dès le début de la vie du composant, et peuvent se matérialiser par des fautes de délais avec le vieillissement du circuit, des fluctuations des tensions d'alimentation ou de la température. Ainsi, un circuit intégré peut être soumis à différents types de fautes durant sa vie. Nous avons vu que ces fautes peuvent être transitoires, intermittentes ou permanentes.

Nous avons confirmé, dans le deuxième chapitre, que des erreurs intermittentes peuvent se manifester avant que le système ne soit défaillant. Avant cela, aucune étude n'avait analysé expérimentalement l'apparition des erreurs intermittentes sur un système embarqué complexe dans une technologie actuelle. Nous avons, en particulier, montré que les erreurs intermittentes peuvent être observées à un niveau système pendant le fonctionnement du circuit intégré. Ainsi, il est possible de définir des méthodes de test en ligne capables de détecter les erreurs intermittentes.

Dans ce sens, la première partie de ce chapitre présentera la notion de tolérance aux fautes et toutes les étapes nécessaires pour la mettre en place. La deuxième partie présentera un état de l'art des techniques de détection en ligne des erreurs. Cela permettra, dans la troisième partie, de comparer les différentes techniques dans le but de déterminer une méthode adaptée à la fois aux erreurs intermittentes et aux architectures multiprocesseurs.

3.1 Critères de sélection

Le but de notre étude est de définir une méthode capable de détecter en ligne des erreurs intermittentes dans une architecture multiprocesseur. Cependant, nous définissons ici un certain nombre de critères auxquels la méthode sélectionnée devra répondre. Tout d'abord, la solution choisie ne devra pas nécessiter la modification des processeurs. En effet, nous considérerons dans notre étude que nous avons peu de connaissances des processeurs et qu'ils peuvent être de plusieurs types différents au sein de l'architecture. De même, l'impact sur les performances et le coût en surface silicium seront des critères déterminants.

Nous avons montré dans les deux chapitres précédents, que le vieillissement varie dans le temps et que le besoin en fiabilité n'est pas le même pour toutes les applications. Ainsi, la méthode que nous voulons déterminer doit pouvoir être adaptée en fonction du vieillissement et en fonction des applications.

Fonctionnalités à apporter :	Peu d'impact sur :
– Capacité à détecter des erreurs intermittentes	– Coût silicium
– Adaptation au vieillissement	– Design (Pas de modification des IP)
– Adéquation application ⇔ niveau de fiabilité	– Consommation
– Gestion en ligne de fiabilité	– Performance
– Évaluation de l'apport en fiabilité	

TABLE 3.1 : *Critères de sélection du procédé de détection en ligne des erreurs.*

3.2 Notions de tolérance aux fautes

3.2.1 Les différentes étapes de la tolérance aux fautes

La notion de tolérance aux fautes regroupe l'ensemble des techniques utilisées pour s'affranchir des fautes dans le circuit et ainsi éviter les erreurs et les défaillances. On trouve aussi

en français le terme de tolérance aux erreurs qui, généralement, se réfère à la même notion de tolérance aux fautes.

La tolérance aux fautes est effectuée en utilisant la redondance matérielle, logicielle, d'information, et/ou temporelle. L'ensemble des étapes utilisées pour rendre un système tolérant aux fautes est le suivant [44] :

- *Masquage*
- *Détection*
- *Isolation*
- *Diagnostic*
- *Reconfiguration/réparation*
- *Rétablissement*

L'ensemble de ces étapes est détaillé dans la suite de cette partie.

Tout système tolérant aux fautes ne comporte pas obligatoirement chacune des étapes précédentes, mais au minimum il doit être constitué d'un mécanisme de détection des erreurs. Pour cette raison, nous concentrerons notre étude sur cette partie, et particulièrement sur la détection en ligne des erreurs.

3.2.2 Masquage

Le masquage consiste à corriger dynamiquement une erreur au cours de l'exécution d'un système. Par exemple, les techniques de NMR [44] se basent sur N modules redondants pour délivrer un résultat correct (voir figure 3.1a). Dans le cas de trois systèmes redondants (Triple-Modular Redundancy), si un des systèmes produit une erreur, alors elle sera corrigée par les deux systèmes sains restant.

De même, les codes correcteurs d'erreurs (ECC) sont des techniques de masquage, comme par exemple les codes Hamming SEC/DED [44]. Dans notre contexte, les codes correcteurs d'erreurs sont principalement utilisés pour protéger des données en mémoire (voir figure 3.1b). Lors de l'enregistrement d'une donnée en mémoire, des informations redondantes sont ajoutées. Ainsi, lors de sa lecture, ces informations redondantes peuvent permettre de détecter ou de corriger, une ou plusieurs erreurs, selon la taille des informations redondantes.

3.2.3 Détection

La détection ou le test des erreurs rassemble plusieurs techniques appliquées à un composant tout au long de sa vie. On peut distinguer les approches *hors ligne* et les approches *en ligne* [45]. Dans la suite de ce chapitre, nous nous intéresserons aux approches de détection en ligne des erreurs, ainsi, il est important de distinguer ces deux types d'approches.

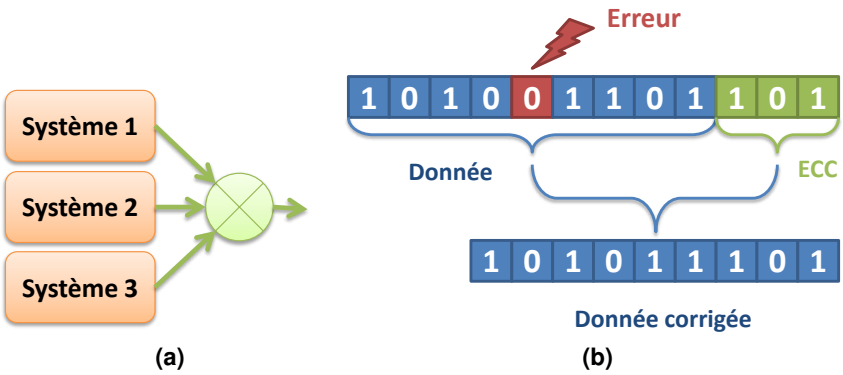


FIGURE 3.1 : *Illustration de méthodes de masquage [44]. (a) Masquage de type TMR : ici, les sorties de trois systèmes sont comparées pour assurer un résultat correct. (b) Code correcteur d'erreurs (ECC) : des informations redondantes sont ajoutées à une donnée pour pouvoir détecter ou corriger une ou plusieurs erreurs.*

Détection hors ligne des erreurs

Exécutés tout au long du flot de fabrication (sur wafer, après découpage des puces, puis après encapsulation), les tests hors ligne permettent d'éliminer de façon rapide et efficace tous les systèmes défectueux comportant des défauts de fabrication. Ces défauts permanents sont révélés par l'application de séquences de détection. Le test consiste à appliquer des vecteurs de test aux entrées du circuit sous test et à comparer les sorties observées à des données de référence pré-calculées par simulation.

Le *test hors ligne* s'effectue alors que le circuit est en dehors de son contexte et que l'on a accès à toutes ses entrées/sorties. Dans ce cas, les vecteurs de test sont envoyés et les résultats analysés à l'aide de testeurs industriels (ATE). Cependant ces testeurs souffrent de nombreux inconvénients [45] :

- une mémoire importante pour stocker les vecteurs de test ;
- un nombre de pointes de test égal au nombre d'entrées/sorties ;
- un temps de test relativement important, par rapport à la vitesse des circuits eux mêmes.

Ainsi, les tests hors ligne sont complétés par des tests en ligne.

Détection en ligne des erreurs

Les techniques de détection en ligne des erreurs sont exécutées pendant que le système est en fonctionnement. Cependant, on distingue deux classes de techniques de détection : *les tests non concurrents* et *les tests concurrents*.

Test en ligne non concurrent Le *test en ligne non concurrent* peut être utilisé au démarrage du système ou à intervalles réguliers durant le cycle de vie du produit.

De la même façon que les *tests hors ligne*, le *test en ligne non concurrent* consiste à appliquer des vecteurs de test aux entrées du circuit sous test et à analyser ses sorties. À la différence que les structures de test sont intégrées dans le composant.

Les techniques de test intégré (BIST) consistent à inclure directement dans le circuit, des fonctions de génération de vecteurs de test (on évite ainsi d'avoir une mémoire dans laquelle tous les vecteurs sont stockés) et d'analyse des résultats. Les dispositifs de test intégrés doivent être capables de générer des vecteurs de test et de comparer les résultats obtenus à ceux attendus suivant le schéma de principe représenté dans la figure 3.2.

Les étapes sont les suivantes [45] :

1. Le *contrôleur de test* met le circuit sous test (CUT) en mode test, c'est à dire que toutes les entrées sont déconnectées pour isoler le circuit pendant la durée du test.
2. Le *générateur de vecteurs de test* envoi sur les entrées du CUT des patterns de test générés de manière pseudo-aléatoire, à l'aide de circuits LFSR par exemple.
3. Les *vecteurs de test* sont propagés à l'intérieur du CUT.
4. Les *sorties* sont comparées avec des signatures de références

On remarque que le circuit doit être dans un mode de test, dans lequel les entrées sont déconnectées, le circuit est donc isolé pendant la durée du test. Ainsi, pour des architectures comportant plusieurs cœurs, un cœur peut être isolé et testé pendant que les autres sont en fonctionnement, c'est pour cela qu'il porte le nom de test en ligne.

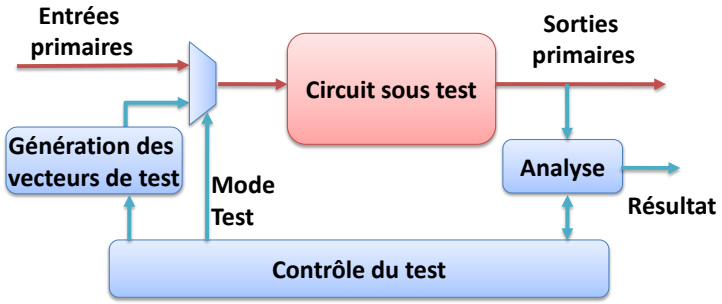


FIGURE 3.2 : *Schéma de principe d'une architecture BIST (Built-in Self-Test).*

Test en ligne concurrent Le *test en ligne concurrent* est exécuté pendant le fonctionnement du circuit. Il est indispensable pour détecter des défauts physiques dus à la fatigue ou à l'usure prématurée du composant, c'est le cas notamment des défauts intermittents et transitoires. Lors d'un test non concurrent, ces types de défauts peuvent échapper au test, le test en ligne a une plus grande probabilité de les détecter.

Les différentes approches de détection en ligne des erreurs

Afin de classer les différentes approches de détection en ligne des erreurs, nous définissons plusieurs catégories :

– Redondance spatiale [46–48] : ces approches utilisent la duplication du système ou de parties du système pour assurer la détection des erreurs.
– Vérification de propriétés [49, 50] : ces approches vérifient certaines propriétés d'exécution pour assurer la validité des résultats produits par le système.
– Utilisation de structures de test [51–53] : ces approches utilisent des structures de test BIST ou des vecteurs de test pour s'assurer que le système n'est pas défaillant.
– Redondance temporelle [54–58] : ces approches utilisent une duplication des données séparées dans le temps pour détecter une erreur.

D'autres approches se basent sur l'observation de symptômes [59] pour détecter des erreurs transitoires. Cependant, nous considérons qu'elles ne sont pas pertinentes pour notre étude.

Ces différentes approches seront détaillées dans la partie suivante.

3.2.4 Isolation

L'isolation des erreurs consiste à éviter leur propagation, une fois qu'elles sont détectées. L'isolation peut être effectuée en utilisant des techniques de détection d'erreurs. Par exemple, les techniques de duplication assurent le confinement des erreurs à l'intérieur de la sphère de redondance.

Notion de sphère de redondance

Le concept de sphère de redondance [47] permet de mieux comprendre les mécanismes de détection d'erreurs et leurs différences. Ce concept s'applique à toutes les méthodes de détection se basant sur une exécution redondante, matérielle, logicielle, temporelle ou spatiale.

La sphère de redondance est l'ensemble logique dans lequel les éléments sont protégés par la détection d'erreur. Pour cela, les données entrant dans la sphère doivent être dupliquées et les données sortantes doivent être comparées avant de sortir de la sphère.

Si une erreur apparaît à l'intérieur de la sphère et se propage jusqu'à ses sorties, alors elle sera détectée, et l'erreur ne pourra pas se propager hors de la sphère de redondance. La figure 3.3a illustre un exemple de sphère de redondance dans laquelle deux microprocesseurs sont protégés, et à l'extérieur de laquelle la mémoire et les entrées/sorties ne sont pas protégées.

Ceci pose les remarques suivantes :
– la sphère de redondance ne couvre pas tous les composants, il faut donc prévoir d'autres moyens pour protéger les éléments extérieurs à la sphère. Dans l'exemple figure 3.3a, la mémoire n'est pas protégée par la sphère de redondance et nécessite donc de mettre en œuvre des techniques telles que ECC ;
– il est important de bien choisir quelles sorties sont à comparer. Si des données critiques ne sont pas comparées, il y a un risque de ne pas détecter les erreurs. Si des sorties non utiles sont comparées, le temps de comparaison et la complexité peuvent être augmentés inutilement ;

– en ce qui concerne les entrées, si elles ne sont pas correctement reproduites ou si le délai est trop important, alors il y a un risque que les différents flux d'exécution diffèrent.

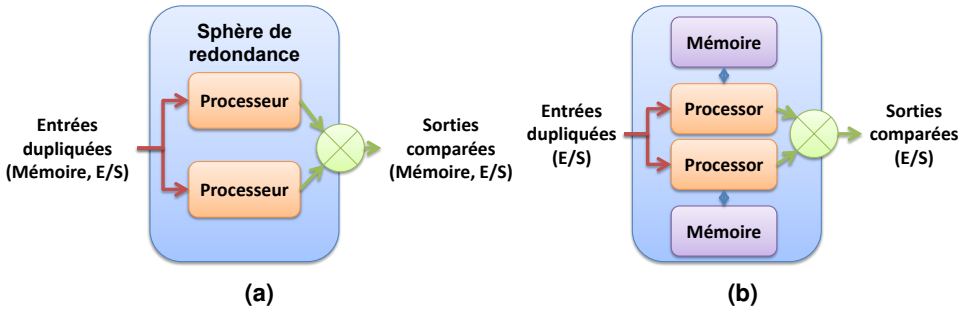


FIGURE 3.3 : *Exemples de sphères de redondance : (a) Excluant la mémoire et les entrées/sorties (b) Incluant la mémoire*

Taille de la sphère de redondance

La taille de la sphère de redondance peut-être très différente d'un système à un autre. Suivant le nombre de composants inclus dans la sphère de redondance, la taille peut-être considérablement augmentée. Par exemple, sur la figure 3.3b la mémoire est incluse dans la sphère de redondance. La mémoire est donc protégée, les accès avec le processeur sont plus rapides mais la taille du système s'en voit considérablement augmentée.

Dans l'industrie, on peut trouver des sphères de redondance relativement différentes. Par exemple dans les *ftServer*[60] de la société Stratus, la mémoire et les microprocesseurs sont inclus dans la sphère de redondance de la même façon que sur la figure 3.3b. Et, au contraire, dans les systèmes *Himalaya*[61] de la société Hewlett-Packard la mémoire n'est pas incluse de la même façon que sur la figure 3.3a. En ce qui concerne le processeur *G5*[62] d'IBM, la sphère de redondance n'inclut qu'une partie du pipeline (les unités de *fetch*, *decode* et *excute*) et exclut les registres internes du processeur.

3.2.5 Diagnostic

L'étape de diagnostic permet de localiser la source de l'erreur. Elle est nécessaire à l'étape suivante, pour rétablir la fonctionnalité du système. La précision de la localisation de l'erreur sera donnée par la méthode de détection employée et les capacités de reconfiguration/réparation de l'architecture. Par exemple, dans une approche en ligne adaptée à une architecture multiprocesseur, le diagnostic consistera à localiser un éventuel processeur défectueux. Dans ce cas, il ne sera pas nécessaire de localiser précisément où l'erreur est apparue à l'intérieur du processeur. Cependant, un diagnostic plus poussé peut être réalisé hors ligne.

3.2.6 Reconfiguration/réparation

Lorsqu'un défaut permanent est détecté et qu'un composant du système n'est plus utilisable, le système doit être capable de se reconfigurer afin de ne plus utiliser le composant défectueux, de le réparer ou de le remplacer.

Dans le cadre d'une architecture multiprocesseur, cette étape est réalisée par la partie contrôle de l'architecture. Par exemple, si un processeur est considéré comme défaillant, il peux être désactivé. Cette désactivation peut être définitive, ou temporaire. En effet, certaines fautes apparaissent avec la température, dans ce cas, la désactivation temporaire du processeur peut suffire à abaisser sa température et ainsi restaurer la fonctionnalité du système.

3.2.7 Rétablissement

Les techniques de rétablissement sont utilisées pour restaurer un système après une erreur intermittente ou transitoire [23]. En effet, si une erreur permanente est détectée dans un module, ce dernier n'est plus utilisable. Dans ce cas, le système peut continuer son exécution sans ledit module, c'est ce que l'on appelle *graceful degradation* ; ou activer un module en double (*spare*). Dans ces deux cas, le système doit être conçu dès l'origine pour compenser cette défaillance, sinon le système entier devient défaillant.

Classification des schémas de restauration

Pour illustrer les mécanismes de restauration, prenons la figure 3.4. L'état initial du système est *00*. Il passe par deux états intermédiaires, *01* et *10*. Puis, en atteignant l'état courant, *11*, une erreur amène le système dans l'état erroné *01*. Pour restaurer le système, trois techniques peuvent être envisagées [23] :

- *reboot* : le système redémarre dans l'état *00* ;
- *forward error recovery* : le système reprend à partir de l'état courant *11* ;
- *backward error recovery* : le système reprend à partir d'un état antérieur à la détection de l'erreur (*01* ou *10*).

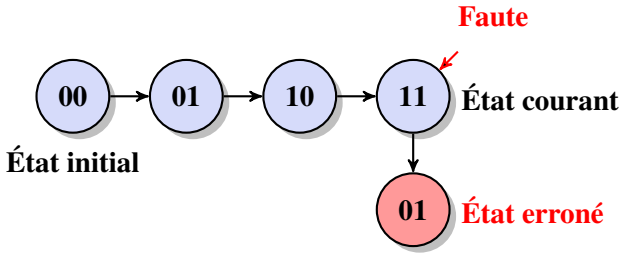


FIGURE 3.4 : *États de restauration après une erreur*

Reboot

Un reboot est le schéma de restauration minimal qui peut être envisagé. Dans le cas d'erreurs intermittentes ou transitoires, en redémarrant, le système retrouve sa fonctionnalité. Si le système peut supporter la latence due au redémarrage du système, c'est une méthode qui est efficace.

Dans le cas d'une erreur permanente, le redémarrage du système ne permettra pas de le restaurer. Une maintenance sera nécessaire.

Forward error recovery

Pour utiliser une technique de *forward error recovery*, le système nécessite de connaitre une version de l'état courant non erroné. Pour cela, le système doit implémenter une technique de redondance pour reconstruire l'état courant. Les techniques de masquage (§ 3.2.2) sont des techniques de *forward error recovery*.

Backward error recovery

Pour implémenter un mécanisme de *backward error recovery*, le système effectue des sauvegardes d'états (*checkpoints*). Si une erreur est détectée, le système reprend à partir de la dernière sauvegarde. Pour que le mécanisme de restauration soit efficace, la fréquence de sauvegarde doit être réfléchie. Si la fréquence est trop faible, les données en mémoire risquent d'être consommées, la restauration devient alors impossible. Si la fréquence est trop grande, le coût engendré par la sauvegarde de l'état du système devient non négligeable. Le processeur IBM série Z [62] implémente un système de *backward error recovery*.

3.3 Les approches de détection en ligne des erreurs

Cette partie présente un ensemble de techniques de détection en ligne des erreurs basées sur des approches matérielles ou logicielles et adaptées à des architectures mono-processeur ou multiprocesseur. Le but de cette partie est, d'une part, de présenter les différentes approches de détection en ligne des erreurs, et d'autre part, de les analyser en fonction de nos critères. Le but de notre étude est de définir une méthode capable de détecter en ligne des erreurs intermittentes dans une architecture multiprocesseur, et nous avons défini au début de ce chapitre les contraintes que doit respecter cette méthode. Ainsi, nous tenterons de comparer chacune des méthodes à partir de ces contraintes.

Comme nous l'avons vu dans la partie précédente, nous nous intéressons aux approches de détection suivantes :

- redondance spatiale [46–48],
- vérification de propriétés [49, 50],
- redondance temporelle [54–58],
- utilisation de structures de test [51–53].

Chaque type d'approche sera détaillé dans une partie différente, et tentera de décrire le type d'erreur visé, la couverture de fautes ainsi que l'impact sur les performances et la surface.

3.3.1 Détection en ligne des erreurs par redondance spatiale

Il existe deux types d'approches de détection utilisant la redondance spatiale. La première approche se base sur la duplication matérielle du système ou parties du système à fiabiliser, c'est le cas du *Locksteping* et du *NMR*. Alors que la deuxième approche se base sur la redondance matérielle déjà existante, c'est le cas du *Redundant Multitheading*.

Lockstepping

Le principe du lockstep [46] est connu et utilisé depuis les années 80 dans les applications à fort besoin de fiabilité. Le lockstep est un procédé de détection par couplage fort, qui met en jeu deux copies redondantes, d'un même programme, synchronisées au cycle près. Un comparateur prend en entrée un ensemble de signaux internes de chaque copie redondante. Si les signaux diffèrent, une erreur est détectée. La méthode du lockstep est une implémentation de DMR.

Couverture de fautes Le principal avantage du lockstep est qu'il ne se focalise pas sur un modèle de faute en particulier. La couverture de fautes est donc maximale pour les erreurs localisées dans le processeur. Le lockstep est capable de détecter les défauts transitoires, intermittents et permanents.

Impact sur les performances Tant qu'il n'y a pas d'erreur, l'impact sur les performances est quasiment nul. En effet, seule la durée de comparaison des sorties est ajoutée, ce qui représente au mieux un cycle d'horloge et au maximum quelques cycles [46].

Impact sur la surface Le coût en surface est généralement élevé mais dépend directement de la sphère de redondance choisie. L'implémentation d'un lockstep nécessite au minimum la duplication d'un processeur, ou de parties de celui-ci mais les différents niveaux de cache peuvent s'ajouter, augmentant ainsi le coût en surface. Dans notre cas, et afin de comparer le coût en surface des différentes techniques, nous considérerons que l'implémentation de cette technique ajoute au minimum 100% de la surface.

Contraintes d'implémentation La mise en place du lockstep nécessite une synchronisation forte pour maintenir les copies dans le même état, ce qui exprime une forte contrainte de conception au niveau de la duplication des entrées et la comparaison des sorties. Si la synchronisation n'est pas faite au niveau cycle, les deux copies peuvent diverger rapidement. Notamment pour les processeurs out-of-order qui peuvent exécuter des instructions de manière spéculative, dans ce cas, les deux copies peuvent prendre des chemins différents.

Ces contraintes peuvent être complexes à mettre en œuvre. Pour les relâcher, il existe une implémentation du lockstep qui ne necéssite pas une synchronisation au cycle près, c'est le *Loose lockstepping* ou Redundant Multitheading (RMT) .

Redundant Multitheading

De la même manière que pour le lockstep, les techniques de Redundant Multitheading (RMT) reposent sur la comparaison des sorties de deux copies redondantes d'un même programme. En ce qui concerne le lockstep, les entrées sont reproduites et les sorties comparées cycle par cycle, alors que pour le RMT ces actions sont réalisées à chaque instruction de *commit*.

Le RMT est un concept qui peut être implémenté sur des architectures intégrant le multi-threading, tel que les processeurs SMT [47] ou CMP [48], avec peu de modifications matérielles. Ces processeurs possèdent plusieurs pipelines d'exécution et permettent l'exécution de plusieurs threads en parallèle. Ici, le processeur est modifié de manière à ce que deux threads exécutent les mêmes opérations. Ainsi, à la différence d'un processeur SMT non modifié, l'implémentation du RMT réduit l'OS à ne voir qu'un seul thread pour deux voies matérielles.

Dans l'industrie, la société Marathon Technologies est la première entreprise à avoir implémentée le concept de RMT dans ses serveurs *Endurance 4000*. Puis récemment Hewlett-Packard est passé des architectures *NonStop* utilisant du lockstep aux architectures *NonStop Advance Architecture* (NSAA) utilisant du RMT.

Contraintes d'implémentation L'implémentation dépend directement du type d'architecture visé, ainsi la technique Simultaneous and Redondant Threading [47] (SRT) repose sur un processeur superscalaire SMT et la technique Chip level Redundant Threading [48] (CRT) repose sur un processeur CMP. Dans l'implémentation qui est faite du SRT, les deux threads sont démarrés avec un certain décalage (de quelques dizaines à quelques centaines de cycles d'horloge), le premier thread est appelé *leading thread* et le deuxième *trailing thread*. Chacun des deux threads produit des résultats, mais ceux-ci sont copiés en mémoire uniquement s'ils sont identiques.

Impact sur des performances En décalant le départ des deux threads, le *trailing thread* bénéficie de tout les défauts de cache du *leading*, c'est ce qui est appelé le *slack fetch*. Cette technique permet de limiter l'impact sur les performances de l'architecture. La dégradation des performances est de 20 à 40% suivant l'application exécutée et par rapport à un système non redondant sans présence d'erreur.

Dans une implémentation de type CRT, le fait que les threads s'exécutent sur des processeurs différents permet de limiter la dépendance de donnée au sein de chaque processeur. Ainsi les performances sont meilleures que pour le SRT. La dégradation des performances est de 10 à 30% suivant l'application exécutée et par rapport à un système non redondant.

Couverture de fautes La couverture de fautes dépend de la sphère de redondance. Ainsi les registres qui ne sont pas dupliqués ne bénéficient pas de protection. Par rapport au lockstep, cette méthode ne bénéficie pas d'une couverture totale des fautes pour l'ensemble du processeur.

Les méthodes de SRT et CRT permettent une détection efficace des erreurs transitoires. En ce qui concerne les erreurs permanentes et intermittentes, l'approche CRT permet une plus grande couverture que l'approche SRT. En effet, les threads sont exécutés sur des processeurs différents, ce qui évite les modes communs.

Synthèse et adaptation à nos contraintes

Nous avons présenté dans cette partie, deux techniques de redondance spatiale : le lockstep et le RMT. Concernant la couverture des erreurs, ces techniques permettent de détecter des erreurs intermittentes. De plus, ces méthodes peuvent être adaptées à des architectures multiprocesseurs, cependant le coût en surface serait très important.

L'implémentation d'un lockstep implique que chaque processeur de l'architecture soit dupliqué, mais elle n'induit pas de dégradation des performances. Cependant, à cause du couplage fort, cette technique nécessite une modification des processeurs, ce que nous voulons éviter.

De même, les méthodes de RMT nécessitent des processeurs multithread pour pouvoir être implémentées. Or, nous considérons une architecture constituée de processeurs non-multithread. Ainsi, l'augmentation de la surface, induite par cette technique, peut être estimée à 40% pour chaque processeur. Cette technique, implique donc un coût en surface inférieur à une technique lockstep, mais nécessite toujours la modification des processeurs.

À la vue des remarques précédentes, les techniques de redondance spatiale ne peuvent répondre à nos contraintes.

3.3.2 Détection en ligne des erreurs par vérification de propriétés d'exécution

Les approches présentées dans cette partie utilisent certaines propriétés d'exécution pour assurer la validité des résultats produits par le système (voir figure 3.5). A la différence des approches précédentes, ces approches ne nécessitent pas de dupliquer les ressources matérielles et sont donc moins coûteuses au niveau de la surface. Les approches présentées sont DIVA [49] et Argus [50].

Dynamic Implementation Verification Architecture

Dynamic Implementation Verification Architecture [49] (DIVA) est un module permettant de détecter des erreurs de calculs ou des erreurs de données dans un processeur out-of-order. Le module DIVA est ajouté à la suite du pipeline du processeur et vérifie le résultat de chaque

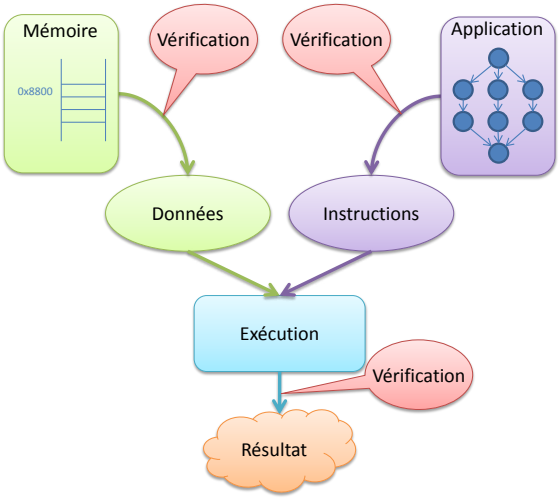


FIGURE 3.5 : *Illustration des méthodes de détection en ligne des erreurs par vérification de propriétés d'exécution.*

exécution. Ce module est composé de deux pipelines de petites tailles, le *CHKcomp pipeline* et le *CHKcomm pipeline* pour chaque flux d'exécution du processeur.

Le *CHKcomp pipeline* vérifie qu'une instruction a été correctement exécutée par le processeur. Pour cela, il est constitué de deux étages, un premier étage recalcule l'opération à partir de l'instruction et des données précédemment utilisées par le processeur. Le deuxième étage compare la donnée ainsi recalculée avec celle obtenue par le processeur. Si les deux résultats correspondent alors le résultat est considéré comme valide, reste à vérifier que les données à l'entrée de l'étage d'exécution sont valides.

Pour cela, le *CHKcomm pipeline* vérifie que le flot de données est correct. Il prend en entrée les opérandes utilisés par le processeur et vérifie qu'ils concordent avec les données présentes dans les registres du processeur. Si les sorties des deux modules DIVA sont corrects alors aucune erreur n'est détectée.

Couverture de fautes Cette méthode est capable de détecter des erreurs permanentes ou des erreurs de design dans le processeur principal, ainsi que des erreurs transitoires ou intermittentes. Ainsi, cette technique pourrait être adaptée à nos contraintes.

Impact sur les performances La dégradation des performances est de 3 à 15% par rapport au processeur seul.

Coût en surface Le coût en surface dépend du processeur utilisé, sur un processeur Alpha 21264, l'implémentation de Diva ne représente que 6% du processeur. Mais au contraire, sur un processeur simple, l'implémentation de Diva peut représenter le double du processeur.

Argus

De la même manière que Diva, Argus [50] se base sur la vérification de propriétés pour assurer la validité de l'exécution.

Un processeur Von Neumann réalise quatre opérations pour obtenir un résultat :

1. choisir la séquence d'instructions à exécuter (*control-flow*) ;
2. exécuter chacune des instructions ;
3. transférer le résultat de chaque instruction, aux instructions ayant des dépendances de données (*data-flow*) ;
4. interagir avec la mémoire.

En vérifiant chacune des quatre propriétés précédentes, Argus assure la détection des erreurs dans le cœur du processeur.

Pour vérifier la première propriété (*control-flow*), Argus compare un graphe de contrôle d'exécution (CFG) statique, calculé lors de la compilation du programme, avec le CFG dynamique reconstruit lors de l'exécution de l'application. Si ceux-ci diffèrent, alors une erreur est détectée.

Pour vérifier la deuxième propriété (exécution), le résultat des opérations des unités fonctionnelles est vérifié en utilisant des modules spécifiques à chaque unité de traitement. Ces modules ne nécessitent pas de ré-exécuter l'opération, mais vérifient la cohérence du résultat [63].

La troisième propriété (*data-flow*) est vérifiée en comparant le graphe de dépendance des données (DFG) statiques en mémoire et le DFG reconstruit lors de l'exécution de l'application. Le DFG est construit à partir de l'historique des états de chaque élément du système (SHS).

Pour finir, les accès mémoire sont protégés en utilisant des codes détecteurs d'erreurs (ECC).

Couverture de fautes Argus ne couvre pas les erreurs sur les entrées/sorties, les exceptions et les interruptions. Mais, couvre respectivement 98% et $98,8\%$ des erreurs transitoires et permanentes. L'efficacité à détecter des erreurs intermittentes n'est pas montrée, cependant, nous pouvons supposer que cette technique en est capable.

Impact sur les performances La dégradation des performances est comprise entre 3% et 10% et varie en fonction de l'application exécutée.

Coût en surface Une implémentation d'Argus a été réalisée dans un processeur OpenRisc 1200 (OR1200), et nécessite une augmentation de surface de $16,6\%$.

Synthèse et adaptation à nos contraintes

Nous avons présenté dans cette partie deux techniques de détection en ligne des erreurs utilisant la vérification de propriétés d'exécution. Concernant la couverture des erreurs, ces tech-

niques peuvent permettre de détecter des erreurs intermittentes, cependant aucune étude ne le certifie. De plus, ces méthodes peuvent être adaptées à des architectures multiprocesseurs, cependant chaque processeur de l'architecture doit être modifié. Or, nous voulons éviter de modifier les processeurs.

Nous constatons, que les deux méthodes induisent une dégradation des performances comprise entre 3% et 10%. Ceci, place ces techniques devant le RMT.

À la vue des remarques précédentes, les techniques de détection en ligne des erreurs utilisant la vérification de propriétés d'exécution, ne peuvent répondre entièrement à nos contraintes.

3.3.3 Détection en ligne des erreurs par redondance temporelle matérielle

Les approches présentées dans cette partie utilisent une duplication des données séparées dans le temps pour détecter une erreur. Ces approches requièrent un support matériel, cependant, celui-ci est très inférieur aux approches de redondance spatiale.

Razor

L'objectif de Razor est d'abaisser au maximum la consommation du processeur. En abaissant la tension d'alimentation, les performances du processeur se dégradent, des retards dans le traitement des données apparaissent et peuvent provoquer des erreurs. Dans ce cadre, le but de Razor est de positionner le processeur dans une zone offrant le meilleur compromis entre la consommation et le taux d'erreurs. En effet, le système Razor peut corriger les erreurs de délais jusqu'à un certain seuil.

Razor[54] permet de détecter et de corriger des erreurs de délais au niveau du pipeline, en effectuant une double capture sur les données entre les étages du pipeline. En présence d'une erreur, la donnée présente lors de la deuxième capture est considérée comme correcte. Deux implémentations sont décrites pour récupérer après cette erreur :

– soit le pipeline est gelé pendant un cycle, le temps de corriger la donnée,
– soit le pipeline est vidé et les instructions ré-exécutées.

Dans tous les cas, la donnée est corrigée avant d'être écrite en mémoire, ce qui évite sa propagation.

Impact sur les performances Sans présence d'erreur, l'utilisation du système Razor n'induit aucune perte de performance. En présence d'une erreur, suivant la méthode d'implémentation choisie, la latence varie entre une période d'horloge et une fois la profondeur du pipeline, en fonction de l'étage où a eut lieu l'erreur.

Couverture de fautes Razor permet de détecter et de corriger uniquement des fautes de délais transitoires. Néanmoins, il peut aussi corriger des erreurs permanentes de délais (dues par exem-

ple au vieillissement), si le délai ajouté reste inférieur à une demie période d'horloge. Dans ce sens, cette méthode est aussi adaptée à la détection des erreurs intermittentes.

Surface L'augmentation de la surface concerne la modification des bascules utilisées pour la double capture, ainsi que la logique de contrôle du pipeline en cas d'erreurs. En pratique, cela ne concerne que le chemin critique du processeur. Ainsi, le surcoût pour le processeur est très faible.

Self-Imposed Temporal Redundancy

En utilisant la redondance temporelle, sur des processeurs out-of-order, le système Self-Imposed Temporal Redundancy[55](SITR) force les unités fonctionnelles à effectuer deux fois le calcul en maintenant les entrées. Si le résultat des deux calculs est différent, le deuxième résultat est considéré comme correct.

Consommation Le fait de maintenir les entrées participe à limiter la consommation d'énergie du test. En effet, si les entrées sont maintenues, il n'y a pas de transition. L'impact sur la puissance dynamique est donc réduit.

Impact sur les performances La dégradation des performances apparait uniquement lors de l'utilisation des unités fonctionnelles. Ainsi, elle dépend directement des applications et peut être estimée entre 10% à 70%.

Coût en surface Pour fonctionner, SITR nécessite de modifier la station de réservation qui envoi les données sur les unités fonctionnelles, ainsi que les unités fonctionnelles elles-mêmes. Le coût en surface est de 1 à 5% suivant le composant (additionneur, multiplieur, ...).

Couverture de fautes D'une part, la détection des erreurs est limitée aux unités fonctionnelles, et d'autre part, elle ne couvre que les erreurs transitoires. En effet, si une faute transitoire apparait pendant le traitement d'une donnée alors elle peut provoquer une erreur. Dans ce cas, la ré-exécution des traitements corrigera l'erreur. Cependant, dans le cas d'une erreur permanente, l'erreur sera toujours présente.

Dans le cas des erreurs intermittentes, la ré-exécution peut être efficace, mais pour cela elle doit être contrôlée. En effet, nous verrons dans le chapitre 4 que le temps entre les deux exécutions détermine l'efficacité de la détection des erreurs intermittentes.

Microarchitecture Based Introspection

Le principe de Microarchitecture Based Introspection[56] est d'utiliser le temps de latence due aux défauts de cache pour ré-exécuter un flux d'opérations. Si les deux exécutions ne ren-

voient pas le même résultat, alors l'opération est exécutée une nouvelle fois pour déterminer le résultat.

Couverture de fautes De même que pour SITR, cette méthode a principalement été conçue pour la détection des erreurs transitoires. Ainsi, pour les mêmes raisons, elle n'est pas efficace pour la détection des erreurs permanentes et très peu pour les erreurs intermittentes.

Coût en surface Afin de modifier le processeur pour mettre en place MBI, il est nécessaire d'ajouter une file de registres supplémentaires, un buffer pour stocker les instructions à exécuter en mode inspection, un mécanisme pour décider de passer en mode inspection et un comparateur pour vérifier les résultats. Ainsi, le coût en surface est principalement donné par la mémoire qui est estimée à 16 ko pour un cache du processeur de 1 Mo.

Impact sur les performances Le coût en performance est inférieur à 15% mais dépend des applications. En particulier, les applications effectuant beaucoup de transferts mémoire ont aussi beaucoup de cache miss et cela laisse plus de temps à MBI pour ré-exécuter des calculs. Dans ce cas, le coût en performance est inférieur à 8%.

Synthèse et adaptation à nos contraintes

Nous avons présenté dans cette partie, trois techniques de détection en ligne des erreurs par redondance temporelle matérielle : Razor, SITR et MBI. Ces trois techniques permettent de détecter des erreurs en comparant des données séparées dans le temps. Ces approches sont principalement destinées à la détection des erreurs transitoires et dans certains cas intermittentes. En effet, si l'erreur est permanente, alors deux exécutions séparées dans le temps donneront le même résultat et l'erreur ne sera pas détectée.

Parmi ces techniques, Razor se détache des autres en proposant une méthode capable de détecter des erreurs de délais avec un impact quasiment nul sur la durée d'exécution des applications. De plus, ces approches peuvent être implémentées dans une architecture multiprocesseur. Cependant, cela nécessite la modification de chaque processeur, ce que nous voulons éviter.

À la vue des remarques précédentes, les techniques de détection en ligne des erreurs par redondance temporelle matérielle ne peuvent répondre à nos contraintes.

3.3.4 Détection en ligne des erreurs par redondance temporelle logicielle

Comme les approches matérielles, les approches logicielles permettent de détecter et corriger les erreurs. Ces approches peuvent être implémentées au niveau de l'application, de l'OS ou de la machine virtuelle. Chaque niveau a ses avantages et ses inconvénients.

Au niveau de l'application, la détection et la restauration sont faciles à intégrer, mais une erreur au niveau de l'OS ne pourra pas être détectée. Au niveau de l'OS, la détection et la restau-

ration sont plus difficiles, puisqu'il faut modifier un OS entier. Au niveau de la couche de virtualisation, celle-ci étant à l'interface entre le matériel et le logiciel, la détection peut bénéficier d'un plus grand taux de couverture. Mais les mécanismes de détection et de restauration inclus au niveau de la couche de virtualisation sont les plus complexes.

Cette partie présente des méthodes implémentant la duplication au niveau de l'application.

Error Detection by Duplicated Instructions

L'approche Error Detection by Duplicated Instructions[57](EDDI) implémente du RMT au niveau logiciel, et contrairement aux approches matérielles, elle ne nécessite qu'un seul contexte hardware.

Les registres et la mémoire sont séparés en deux pour permettre deux copies redondantes des données. Ainsi, la mémoire est dans la sphère de reproduction, mais la quantité de mémoire disponible pour les applications est divisée par deux.

L'approche EDDI repose sur le compilateur qui duplique chacune des instructions du programme et ajoute des instructions de comparaison avant chaque *store* (voir tableau 3.2). Néanmoins, après la comparaison les données sont dupliquées dans la mémoire. Cela permet de détecter une éventuelle faute transitoire ayant affecté la mémoire.

Couverture de fautes La méthode EDDI a été développée pour détecter des erreurs transitoires. En effet, dans le cas d'erreurs permanentes ou intermittentes, la ré-exécution d'une instruction donnera le même résultat, l'erreur ne sera pas détectée.

Coût en surface Pour implémenter EDDI, il n'est pas nécessaire de modifier l'architecture du processeur, seule l'application nécessite d'être modifiée. Cependant, le coût en surface n'est pas nul. En effet, il faut considérer l'ajout de mémoire engendrée par la duplication.

Impact sur les performances La dégradation des performances est de 36% à 111% sur un processeur super-scalaire avec deux contextes hardware. Nous pouvons supposer qu'elle serait supérieure pour un processeur simple ayant un seul contexte.

Sans EDDI	Avec EDDI
LOAD R12 = [R11]	LOAD R12 = [R11]
	LOAD R22 = [R21]
ADD R11 = R12 + R13	ADD R11 = R12 + R13
	ADD R21 = R22 + R23
	COMPARE R11,R21

TABLE 3.2 : *Implémentation de EDDI[57]*

Software Implemented Fault Tolerance et CompileR Assisted Fault Tolerance

Afin d'améliorer la technique précédente Software Implemented Fault Tolerance[58](SWIFT) et CompileR Assisted Fault Tolerance[64](CRAFT) ont été développées. La principale différence avec EDDI est que la mémoire n'est plus dans la sphère de reproduction. Il n'y a donc plus de perte de mémoire. En effet, les auteurs supposent que la mémoire est protégée par ECC. Ainsi, seuls les registres internes doivent être dupliqués.

Pour SWIFT, l'implémentation reste la même que pour EDDI, c'est à dire que le compilateur duplique les instructions de l'application et ajoute des étapes de comparaison.

Dans le cas de CRAFT, la comparaison lors des *store* est faite en matériel. Cette solution est donc hybride : c'est une solution logicielle bénéficiant d'accélérateurs matériels spécifiques.

Couverture de fautes Les techniques SWIFT et CRAFT s'appuyant sur EDDI, elles sont adaptées principalement à la détection des erreurs transitoires. De même, elles possèdent les mêmes inconvénients concernant la détection des erreurs permanentes et intermittentes.

Coût en surface Concernant SWIFT, comme EDDI, il n'est pas nécessaire de modifier l'architecture du processeur, seule l'application nécessite d'être modifiée. Cependant, l'implémentant de SWIFT implique la duplication des registres internes et nécessite une mémoire protégée par ECC.

Concernant CRAFT, le processeur nécessite d'être modifié pour ajouter la comparaison matérielle des registres. Cependant, les structures ajoutées sont extrêmement simples, ainsi, le coût en surface est très faible.

Impact sur les performances La dégradation des performances induite par SWIFT et CRAFT dépend des applications ; elle est comprise entre 10% et 85% pour SWIFT et entre 10% et 70% pour CRAFT. De plus, grâce à la comparaison matérielle, CRAFT induit en moyenne moins de 10% de dégradations par rapport à SWIFT. Ces estimations ont été réalisées sur un processeur super-scalaire Intel Itanium 2. Nous pouvons supposer qu'elles seraient supérieures pour un processeur simple ayant un seul contexte hardware.

Inst. type	NOFT	SWIFT	CRAFT
STORE	st [r1] = r2	br faultDet, r1 != r1' br faultDet, r2 != r2' st [r1] = r2	st1 [r1] = r2 st2 [r1'] = r2'
LOAD	ld r1 = [r2]	br faultDet, r2 != r2' ld r1 = [r2] mov r1' = r1	ld1 r1 = [r2] ld2 r1' = [r2']

TABLE 3.3 : *Implémentation de SWIFT[58] et CRAFT[64]*

Synthèse et adaptation à nos contraintes

Dans cette partie, nous avons présenté trois techniques de détection en ligne des erreurs par redondance temporelle logicielle : EDDI, SWIFT et CRAFT. Le principal avantage de ces techniques est qu'elles ne nécessitent pas de modification des processeurs (sauf pour CRAFT). Seule l'application doit être modifiée, et cette étape est réalisée automatiquement par le compilateur. Ainsi, ces techniques peuvent être utilisées sur des architectures multiprocesseur.

Ces techniques ont été conçues pour détecter des erreurs transitoires, ainsi, elles ne sont pas adaptées à nos contraintes. De plus, la dégradation des performances des applications peut être très importante et dépasser 110%.

À la vue des remarques précédentes, les techniques de détection en ligne des erreurs par redondance temporelle logicielles ne peuvent répondre à nos contraintes.

3.3.5 Détection en ligne des erreurs par utilisation de structures de test

Les approches présentées dans cette section utilisent des méthodes de type test (généralement à base de BIST, voir §3.2.3) pour détecter des défauts. Un certain nombre de vecteurs de test sont injectés sur les entrées du système à tester et l'analyse des sorties permet de détecter les défauts éventuels. Les approches présentées dans cette partie sont Bulletproof [51], CASP[53] et le Software-Based Self-Test [52].

Bulletproof

Le Bulletproof [51] pipeline utilise du BIST (voir §3.2.3) pour détecter et diagnostiquer les erreurs. Puis avec des points de sauvegarde, il permet de restaurer le système en cas d'erreurs.

Une structure de BIST est insérée sur chaque étage du pipeline. Bulletproof utilise les temps d'inactivité du processeur afin de limiter l'impact sur les performances. Ainsi, l'ensemble du test n'est pas réalisé en une fois, mais il est étalé au court des différents temps d'inactivité du processeur.

Couverture de fautes Bulletproof est capable de détecter 89% des erreurs permanentes mais n'est pas capable de détecter les erreurs transitoires. Cependant, il peut être utilisé pour détecter les erreurs intermittentes, si les tests sont exécutés périodiquement (voir chapitre 4).

Impact sur les performances La dégradation des performances est de 4 à 18% en fonction de l'application exécutée. En effet, même si Bulletproof utilise les temps d'inactivité du processeur, le test complet du processeur doit être réalisé en accord avec les périodes de checkpoint. Si les périodes d'inactivité sont trop faibles, alors le test est forcé, ce qui augmente le coût en performance.

Coût en surface Le coût en surface sur un processeur VLIW quatre voies est de 9,6% sans compter les caches. Dans ce cas, les patterns de test utilisés pour le BIST sont mutualisés (car identiques pour chaque voie), ainsi, le coût en surface serait plus important pour un processeur mono voie. En effet, le coût en surface doit comprendre la mémoire utilisée pour stocker les vecteurs de test.

CASP

Concurrent Autonomous Chip Self-Test Using Stored Test Patterns (CASP)[53] est une solution dédiée à des architectures multiprocesseurs. Elle consiste à tester les processeurs à partir des éléments de DFT déjà présents autour du processeur. Ces éléments sont utilisés pour effectuer les tests de production et permettent de tester l'ensemble du processeur. Les vecteurs de test sont stockés dans une mémoire externe, ce qui permet de mettre à jour les tests au cours de la vie du composant.

Néanmoins, CASP nécessite d'avoir accès aux éléments de DFT (BIST par exemple) de l'architecture et de pouvoir isoler matériellement le processeur pour y appliquer les tests. Dans ce cas, l'isolation est effectuée par le contrôle de l'architecture.

Afin d'améliorer les performances de CASP et de permettre son utilisation dans une architecture massivement parallèle, une autre solution a été proposée : Virtualization-Assisted Concurrent Autonomous Self-Test (VAST)[65]. VAST permet un meilleur ordonnancement des tests en utilisant la virtualisation.

Couverture de fautes L'avantage d'utiliser les ressources de DFT déjà présentes dans l'architecture est que les tests peuvent être appliqués à l'ensemble du processeur sans restriction. Ainsi, la couverture de fautes peut être supérieure aux approches précédentes. Les auteurs rapportent pour le test d'un processeur OpenSPARC T1, un taux de couverture de $99,49\%$ pour les fautes de collage et $95,96\%$ pour les fautes de transition. Comme toute méthode de test, les erreurs transitoires ne peuvent pas être détectées par cette technique. De la même façon que pour Bulletproof, cette méthode peut être utilisée pour détecter les erreurs intermittentes si le test est effectué périodiquement.

Impact sur les performances et coût en surface Si les patterns sont contenus dans une mémoire flash externe le temps de test est d'environ $1,2$ secondes (1s pour le transfert depuis la mémoire externe et $0,2$s pour l'exécution des tests). Cependant, les auteurs se reposent sur un processeur redondant. Dans ce cas, l'impact sur les performances est très faible mais nécessite un processeur supplémentaire, ce qui est à prendre en compte dans le coût en surface et le contrôle de l'architecture.

Si l'on ne considère aucun processeur redondant, le coût en surface est quasiment nul. Cependant, dans tous les cas, il faut prendre en compte la mémoire nécessaire pour stocker les patterns de test. La taille de la mémoire dépend du nombre de vecteur de test et donc du taux de couverture voulu, les auteurs ont identifié une taille de $5,3$ Mo avec une compression de 10x.

Software-Based Self-Test

Le Software-Based Self-Test (SBST) [52] consiste à appliquer des patterns de tests aux différents composants du processeur en utilisant des instructions du processeur lui-même. Les données utilisées par le processeur servent alors de patterns de tests et les instructions servent à cibler un composant particulier. On peut remarquer que le fonctionnement est celui d'un BIST, à la différence que le processeur ne nécessite pas d'être physiquement découplé du système.

Trois étapes sont nécessaires pour appliquer le test :
- *Préparation du test :* dans l'exemple du tableau 3.4, les données *X* et *Y* sont chargées respectivement dans les registres *r1* et *r2*, et seront utilisées comme patterns de test.
- *Application du test :* les données présentes dans les registres *r1* et *r2* sont appliquées sur les entrées du composant *additionneur* (add).
- *Récupération des réponses :* la sortie de l'*additionneur* est enregistrée pour analyse.

Couverture de fautes La couverture de fautes est donnée par les vecteurs de test utilisés. Ainsi, la méthode SBST peut être utilisée pour produire un test fonctionnel ou structurel suivant la description des composants disponibles.

Par exemple, si le concepteur des tests possède une description au niveau porte ou au niveau RTL de son processeur, alors celui-ci peut générer des vecteurs de tests à partir d'outils industriels. Les vecteurs ainsi générés permettent d'obtenir une très bonne couverture de fautes (entre 90% et 98% suivant le processeur et pour des fautes de collage [52]) avec un minimum de vecteurs. Dans ce cas, les vecteurs doivent être stockés en mémoire, ce qui peut représenter plusieurs méga-octets.

Si au contraire, le concepteur des tests ne dispose que du jeu d'instruction du processeur (ISA) alors seule une approche fonctionnelle est possible. Dans ce cas, les vecteurs sont générés de façon aléatoire, soit en ligne, soit hors ligne. Dans cette approche la couverture de fautes est rarement supérieure à 90% [52] et demande plus de vecteurs que l'approche structurelle.

Comme toute méthode de test, les erreurs transitoires ne peuvent pas être détectées par cette technique. De la même façon que Bulletproof et CASP, cette méthode peut être utilisée pour détecter les erreurs intermittentes si le test est effectué périodiquement.

Impact sur les performances Le coût en performance est directement dépendant du nombre de vecteurs de test utilisés. Pour un processeur OpenRISC 1200 le temps de test pour obtenir une couverture des fautes de collage de 95% est de 56 716 cycles d'horloge [66].

Pour une utilisation en ligne de méthodes de SBST, il faut ajouter le temps nécessaire à l'isolement du processeur à tester. En effet, le test étant une tâche logicielle, aucune autre application ne peut s'exécuter en parallèle. Ainsi, l'application doit être stoppée le temps du test ou déplacée sur un autre processeur. En particulier dans une architecture multiprocesseur, il peut être nécessaire de prévoir la migration des tâches présentes sur le processeur.

Coût en surface Les méthodes de SBST étant logicielles, il n'y a pas de coût supplémentaire en surface. Cependant, il faut prévoir la taille de la mémoire en fonction des vecteurs de tests à stocker.

load r1, X	– Préparation du test
load r2, Y	– Préparation du test
add r3, r1, r2	– Application du test
store r3, Z	– Récupération de la réponse

(a)

Memory
Self-test code
Self-test data
Self-test responses

(b)

TABLE 3.4 : *Fonctionnement du SBST. (a) Exemple d'algorithme. (b) Exemple de plan mémoire.*

Synthèse et adaptation à nos contraintes

Nous avons présenté dans cette partie trois techniques de détection en ligne des erreurs par l'utilisation de structures de tests : Bulletproof, CAST et SBST. Toutes ces techniques permettent de détecter les erreurs permanentes et peuvent permettre de détecter les erreurs intermittentes, cependant, ce dernier point reste à étudier.

Parmi ces techniques, Bulletproof ne correspond pas à nos contraintes car nécessite la modification des processeurs. Néanmoins, CASP et SBST correspondent à tous nos critères. Cependant, dans le cas de CASP, l'accès aux éléments de DFT est nécessaire, ce qui peut être réalisé par une modification de l'architecture. De plus, dans ces cas, des éléments facilitant la migration des tâches au sein de l'architecture peuvent limiter l'impact sur les performances.

À la vue des remarques précédentes, les techniques CASP et SBST de détection en ligne des erreurs par l'utilisation de structures de tests répondent à nos contraintes. Pour cela, elles doivent être utilisées au sein d'un test périodique. Or, une étude est nécessaire pour évaluer l'efficacité des tests.

3.4 Conclusion

Ce chapitre a présenté dans une première partie la notion de tolérance aux erreurs. Cela nous a permis de concentrer nos efforts sur la recherche d'une méthode de détection des erreurs. Dans ce sens, ce chapitre établi un inventaire des différentes méthodes de détection en ligne des erreurs utilisées dans l'industrie ou décrites dans la littérature.

La comparaison de ces techniques avec nos critères de recherche met en évidence une méthode basée sur le Software-Based Self-Test (SBST) ou sur l'utilisation de structures de test BIST. En effet, nous recherchons une méthode capable de détecter en ligne des erreurs intermittentes

dans une architecture multiprocesseur. De plus, ces méthodes ne nécessitent pas la modification des processeurs. Elles permettent de détecter les erreurs permanentes mais pas les erreurs transitoires. Concernant les erreurs intermittentes, l'efficacité reste à déterminer avec l'utilisation d'un test périodique. En effet, peu d'études visent ces erreurs, et sans une étude approfondie, il est difficile de se positionner.

Pour cela, le prochain chapitre étudiera la probabilité de détecter des erreurs intermittentes en utilisant un test périodique.

CHAPITRE 4

Définition d'une méthode de test en ligne multiprocesseur

Sommaire

NOUS AVONS VU dans le chapitre précédent que des méthodes basées sur du Software-Based Self-Test (SBST) ou sur l'utilisation de structures BIST pourraient convenir pour détecter, en ligne, des erreurs intermittentes. Cependant, nous devons évaluer l'efficacité de ces méthodes dans une architecture multiprocesseur. Nous savons que l'utilisation de ces

méthodes, au sein d'un *test périodique*, peut permettre la détection des erreurs intermittentes. Or, les contraintes temps-réel des applications et l'implémentation multiprocesseur, peuvent compromettre l'efficacité du test périodique.

Notre objectif est de définir une méthode qui s'adapte au mieux à la fiabilité des applications et au vieillissement de l'architecture, tout en ayant un impact limité sur les performances. Ainsi, nous envisageons que les applications aient une priorité supérieure à celle du test et nous proposons un test pseudo-périodique. Ce point nous différencie des approches de la littérature qui s'appuient sur un test périodique à période fixe. En effet, ces approches considèrent que le test doit être prioritaire et que les applications doivent s'adapter au test. Nous démontrerons, dans ce chapitre, que notre approche de test pseudo-périodique est adaptée aux architectures multiprocesseur. Nous montrerons, que nous pouvons garantir la probabilité de détection du test périodique même en cas de variations de sa période.

Pour arriver à ce résultat, nous commencerons par présenter notre approche de test pseudo-périodique et les contraintes apportées par une implémentation dans une architecture multiprocesseur. Nous verrons, ensuite, comment les erreurs intermittentes peuvent être modélisées par un modèle de Markov. Cela permettra de définir une métrique : la probabilité de détection du test. Nous utiliserons cette métrique pour comparer l'efficacité des tests dans différentes conditions. Dans ce sens, nous étudierons l'évolution de la probabilité de détection en fonction de variations de la période de test. Cette partie permettra de conclure sur notre étude et de valider notre approche.

4.1 Présentation de notre approche de test en ligne des erreurs intermittentes

Principe

Les tests périodiques sont utilisés depuis plusieurs années pour détecter des erreurs intermittentes et permanentes dans les architectures mono-processeur [67, 68]. Le principe est de tester périodiquement le processeur avec une période T (voir figure 4.1). Pour cela, l'application doit être stoppée, le temps du test.

Deux paramètres sont importants dans un test périodique : la période de test et le test en lui-même.

La période de test : elle définie la probabilité de détection du test et son coût. En effet, plus la période est faible et plus la probabilité de détecter une erreur intermittente est importante. À l'inverse, plus la période est faible et plus le test a un impact important sur la durée de l'application. Ainsi, la période doit être adaptée aux erreurs. Nous verrons dans la partie 4.2, comment déterminer la période à partir d'une modélisation des erreurs intermittentes, pour atteindre une probabilité de détection donnée.

Le test : il définit la manière dont les erreurs sont détectées et avec quelle qualité. Les différentes méthodes permettant de détecter des erreurs en ligne ont été étudiées dans le chapitre

précédent. Nous avons sélectionné des méthodes utilisant des structures de test, comme les méthodes de Software-Based Self-Test (SBST) ou des méthodes utilisant des structures de BIST. Pour fonctionner, des vecteurs de test sont injectés sur les entrées du système à tester et l'analyse des sorties permet de détecter les erreurs éventuelles. La qualité du test dépendant en partie du nombre de vecteurs de test utilisé, ce qui a un impact sur la durée du test. Dans le cadre de ce chapitre nous considérerons que la couverture du test est proche de 100%. Si une erreur apparait pendant un test, alors elle sera détectée.

Si la détection est effectuée à l'aide de méthodes de SBST, alors les tâches de test sont logicielles. Si la détection est effectuée par des méthodes utilisant des structures de BIST, alors le test est matériel.

Récemment plusieurs études ont défini un test en ligne périodique à base de méthodes de SBST pour la détection des erreurs intermittentes [29, 69]. Ces études ont permis de montrer l'efficacité des méthodes de SBST dans la détection de ces erreurs. Cependant, aujourd'hui, aucune implémentation n'a été proposée pour une architecture multiprocesseur. Apostolakis et al. [70] ont proposé d'utiliser des méthodes de SBST dans des architectures multiprocesseur, mais seulement dans le cadre d'un test de production. Ainsi, les contraintes ne sont pas les mêmes que pour du test en ligne. De même, Li et al. [71] ont proposés un test périodique à base de SBST dans une architecture multiprocesseur. Cependant, les erreurs visées sont permanentes. Dans ce cas, les contraintes sur la période de test sont très différentes.

FIGURE 4.1 : *Illustration du test périodique*

Contraintes d'implémentation et objectif de l'étude

Si le principe est simple à mettre en œuvre dans une architecture monoprocesseur, ce n'est pas le cas dans une architecture multiprocesseur. Comme nous l'avons vu précédemment, le test doit être ordonnancé avec les applications. L'étude présentée par Li et al. [71] se rapproche de nos objectifs, cependant, les erreurs visées ne sont pas intermittentes et le test est prioritaire. C'est à dire que la période de test est fixe, ce qui impose de fortes contraintes pour les applications.

A l'inverse, notre objectif est de limiter au maximum l'impact du test sur les applications. Ainsi, le test n'est pas nécessairement prioritaire, les périodes de test peuvent ne pas être respectées. Dans ce cas, nous nommerons ce type de test : pseudo-périodique. Les figures 4.2a et 4.2b illustrent ce phénomène.

La figure 4.2a, présente un exemple d'ordonnancement de différentes tâches sur une architecture composée de trois processeurs. Après ajout des tâches de test, si ce dernier n'est pas prioritaire, alors le cas présenté sur la figure 4.2b peut apparaitre. Dans ce cas, le test devient pseudo-périodique et la période de test n'est pas respectée sur le processeur P2, ce qui peut nuire à la détection des erreurs.

L'objectif de ce chapitre est d'analyser l'impact des variations de la période de test sur la détection des erreurs. Nous confirmerons ainsi qu'un test pseudo-périodique peut être utilisé et que le test ne doit pas nécessairement être prioritaire. Cela nous permettra, dans le chapitre suivant, de nous intéresser à l'ordonnancement des tests dans une architecture multiprocesseur.

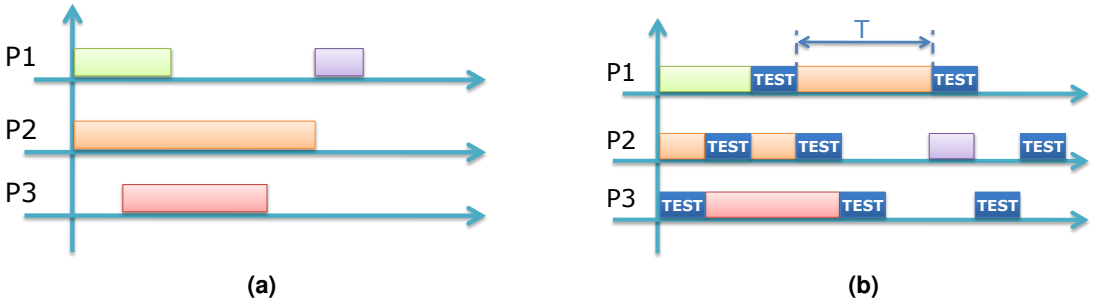


FIGURE 4.2 : *Exemple de test périodique multiprocesseur. (a) Exemple d'ordonnancement de différentes tâches sur une architecture de trois processeurs. (b) Ajout de tâches de test sur un ordonnancement existant. On remarque, dans ce cas, que la période de test n'est pas respectée sur le processeur P2.*

4.2 Étude théorique

Cette partie présente tout d'abord le modèle statistique avec lequel nous allons modéliser les erreurs intermittentes. Cette modélisation permettra de calculer la probabilité de détection du test périodique à tout instant en fonction du type d'erreur à détecter et des instants de test.

4.2.1 Caractéristiques des erreurs intermittentes

Dans le chapitre 2, nous avons étudié l'apparition des erreurs intermittentes. Nous avons observé que ces erreurs apparaissent sous la forme de *bursts*. Au cours de ces *bursts*, les erreurs peuvent être caractérisées par un temps moyen entre deux erreurs (TMEDE) et un temps moyen en erreur (TMEE). La figure 4.3 rappelle ces caractéristiques.

Nous avons en particulier observé dans la partie 2.5.6 comment évoluent ces paramètres en fonction du vieillissement. Nous avons conclu, pour notre cas d'étude, que le TMEDE diminue alors que le TMEE augmente avec le vieillissement.

La prochaine partie utilise ces caractéristiques pour modéliser les erreurs intermittentes.

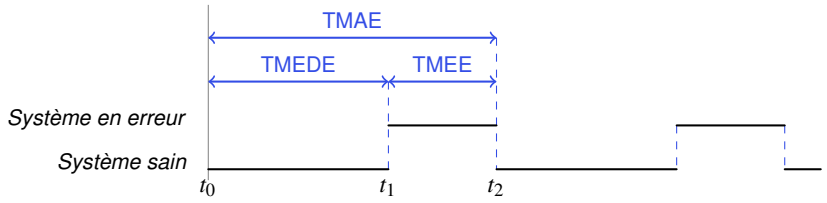


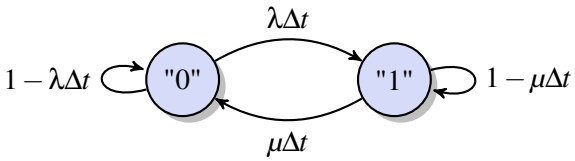
FIGURE 4.3 : *Les bursts d'erreurs intermittentes peuvent être caractérisés par les paramètres suivants : TMEDE = Temps moyen entre deux erreurs, TMEE = Temps moyen en erreur, TMAE = Temps moyen avant erreur.*

4.2.2 Modélisation des erreurs intermittentes

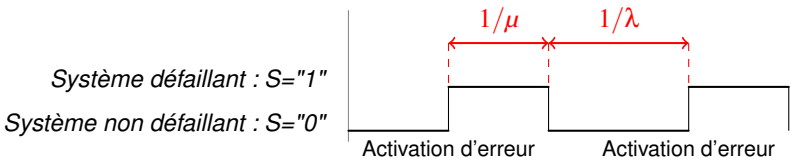
Introduction au modèle de Markov

Pour décrire l'évolution des défauts intermittents, on considère un modèle de Markov à deux états à paramètres continus [29, 72, 73] (voir figure 4.4a). Le modèle de Markov permet de prédire le futur en ayant une connaissance du présent, et une connaissance très limitée du passé (les données du passé n'affinent pas la prédiction). Ceci va nous permettre d'analyser le comportement du système dans le temps en ayant pour seules informations : le temps moyen entre deux erreurs intermittentes et leur durée moyenne d'activation (voir figure 4.4b) ; ou plus exactement les probabilités qu'une erreur s'active et se désactive au cours du temps.

Soit λ et μ, respectivement, les taux de passage de l'état opérationnel ($S = 0$) à l'état défaillant ($S = 1$) et inversement, et $S(t)$ l'état du système à l'instant t. Pour ce modèle, le temps moyen passé dans chacun des états opérationnels et défaillants est défini par une distribution exponentielle de paramètres $1/\lambda$ et $1/\mu$. Ces deux paramètres sont considérés comme constants pour notre étude. En effet, $1/\lambda$ correspond au TMEDE et $1/\mu$ au TMEE définis dans la partie 2.5.6. Ainsi, dans la réalité, λ et μ évoluent avec le vieillissement du système.



(a)



(b)

FIGURE 4.4 : *Illustration du modèle de Markov utilisé et de ses paramètres. (a) Modèle de Markov à paramètres continus utilisé pour modéliser l'état du système en présence d'erreurs intermittentes. (b) Caractéristiques des erreurs intermittentes. $1/\mu$ = durée moyenne des erreurs et $1/\lambda$ = temps moyen sans erreur.*

Probabilités de changement d'états

Le modèle de Markov permet de calculer à chaque instant la probabilité de changer d'état ou de se maintenir dans le même état. On définit $P_{i,j}(t)$ la probabilité de passer d'un état i à t_0 à un état j à t_0+t. Ainsi, $P_{0,1}(t)$ est la probabilité de passer d'un état opérationnel à un état défaillant au bout d'un temps t. De même, $P_{0,0}(t)$ est la probabilité de rester dans l'état opérationnel après un temps t. Voici ces équations :

$$\begin{aligned} P_{0,1}(t) &= \frac{\lambda}{\lambda+\mu}(1-e^{-(\lambda+\mu)t}) \\ P_{0,0}(t) &= 1-P_{0,1}(t) = \frac{\mu}{\lambda+\mu}+\frac{\lambda}{\lambda+\mu}e^{-(\lambda+\mu)t} \\ P_{1,0}(t) &= \frac{\mu}{\lambda+\mu}(1-e^{-(\lambda+\mu)t}) \\ P_{1,1}(t) &= 1-P_{1,0}(t) \end{aligned}$$

De plus, on définit les états stationnaires du système : π_0 et π_1. Ils correspondent à la probabilité d'être dans un des deux états à un temps t quelconque. Ces états sont obtenus en observant le système sur un temps infini, on a donc :

$$\begin{aligned} \pi_1 &= P_{0,1}(\infty) = \frac{\lambda}{\lambda+\mu} \\ \pi_0 &= P_{1,0}(\infty) = \frac{\mu}{\lambda+\mu} = 1-\frac{\lambda}{\lambda+\mu} \end{aligned}$$

4.2.3 Probabilités de détection des erreurs intermittentes

Pour détecter les erreurs intermittentes, nous utilisons un test pseudo-périodique. Ainsi, nous considérons pour cette étude que le temps entre deux tests peut différer de la période de test. Et cela contrairement aux études de la littérature [29, 72, 73]. En effet, ces études considèrent un test strictement périodique. Dans ce cas, le temps entre deux tests est toujours égal à la période de test.

Nous définissons ΔT_n comme le temps séparant les tests $n-1$ et n. Alors, en utilisant le modèle de probabilité précédent, on peut définir $P_d(n)$, la probabilité de détecter un défaut après n tests. $P_d(n)$ est égale à la probabilité que le système reste dans un état opérationnel ($P_{0,0}$) pendant les $n-1$ tests et qu'il soit défaillant lors du $n^{\text{ième}}$ test. On a donc :

$$\begin{aligned} \forall n &\in \mathbb{N}^* \\ P_d(n) &= P_{0,0}(\Delta T_1)\dots P_{0,0}(\Delta T_{n-1}).P_{0,1}(\Delta T_n) \\ &= (1-P_{0,1}(\Delta T_1))\dots(1-P_{0,1}(\Delta T_{n-1})).P_{0,1}(\Delta T_n) \end{aligned} \tag{4.1}$$

A partir de la formule précédente, nous pouvons définir $P_{fd}(N)$ la probabilité d'avoir détecté un défaut après n'importe lequel des N tests. $P_{fd}(N)$ est égale à la probabilité de détecter une

erreur lors du test 1 ou lors du test 2 ... ou du test N. Sachant que toutes les probabilités sont disjointes, on peut écrire :

$$P_{fd}(N) = \sum_{n=1}^{N} P_d(n) = 1 - \prod_{n=1}^{N} P_{0,0}(\Delta T_n) \tag{4.2}$$

La formule précédente doit être contrastée avec la probabilité d'occurrence des erreurs. Soit $q(t)$ la probabilité d'occurrence d'une erreur après un temps t. On définit E_1 l'événement qu'une faute soit inactive à $t = 0$ et E_2 l'événement que la faute reste inactive sur l'intervalle $[0,t]$. Alors en passant par le complément on peut écrire :

$$1 - q(t) = P\{E_1 \cap E_2\} = P\{E_1\}.P\{E_2|E_1\}$$

Or la probabilité de E_1 est égale à la probabilité de présence dans l'état opérationnel donc $P\{E_1\} = \pi_0$. De plus la probabilité de $E_2|E_1$ est directement donnée par $P\{E_2|E_1\} = e^{-\lambda t}$. On a donc $1 - q(t) = \pi_0 e^{-\lambda t}$. On peut donc écrire que la probabilité qu'une faute se déclare sur la période $[0,t]$ est égale à :

$$q(t) = 1 - \pi_0 e^{-\lambda t} \tag{4.3}$$

Si on appelle T_N l'instant du test N, alors, à partir des équations (4.2) et (4.3) on peut déterminer la probabilité qu'une faute se déclare sur l'intervalle $[0,T_N]$ mais soit détectée lors de n'importe lequel des N tests. Soit $d(N)$ cette probabilité :

$$\begin{aligned} d(N) &= q(T_N).P_{fd}(N) \\ &= (1 - \pi_0 e^{-\lambda T_N}).(1 - \prod_{n=1}^{N} P_{0,0}(\Delta T_n)) \end{aligned} \tag{4.4}$$

Cas particulier : test périodique à période fixe

Dans le cas où le test est strictement périodique, l'intervalle entre deux tests est constant et égal à T. Dans ce cas, les équations (4.1), (4.2) et (4.4) peuvent être notablement simplifiées :

$$\forall n \in [1;N] \quad \Delta T_n = T$$

$$P_d(n) = P_{0,0}(T)^{n-1}.P_{0,1}(T) \tag{4.5}$$

$$P_{fd}(N) = \sum_{n=1}^{N} P_d(n) = 1 - P_{0,0}(T)^N \tag{4.6}$$

$$d(N) = q(N.T).P_{fd}(N) = (1 - \pi_0 e^{-\lambda.N.T}).(1 - P_{0,0}(T)^N) \tag{4.7}$$

Latence de détection

La latence de détection $L(T)$ est le temps moyen mis pour détecter une erreur [74]. Ce temps est un multiple de la période de détection T. Si l'erreur est détectée au bout du premier test, alors la latence de détection sera égale à $t_0 + T$. Si elle est détectée au bout du deuxième test, alors la latence de détection sera égale à $t_0 + 2T$, etc.

On a donc :

$$\begin{aligned} L(T) &= T.P_d(1) + 2.T.P_d(2) + 3.T.P_d(3) + \dots \\ &= T.P_{0,1}(T) + 2.T.P_{0,0}(T).P_{0,1}(T) + 3.T.P_{0,0}(T)^3.P_{0,1}(T) + \dots \\ &= \sum_{n=0}^{\infty} (n+1).T.P_{0,0}(T)^n.P_{0,1}(T) \\ &= T.P_{0,1}(T).\sum_{n=0}^{\infty} (n+1)P_{0,0}(T)^n \end{aligned}$$

Or on démontre :

$$\sum_{n=0}^{\infty} (n+1)x^n = \frac{1}{(1-x)^2}$$

D'où :

$$\begin{aligned} L(T) &= \frac{T.P_{0,1}(T)}{(1-P_{0,0}(T))^2} \\ L(T) &= \frac{T.P_{0,1}(T)}{P_{0,1}(T)^2} \\ L(T) &= \frac{T}{P_{0,1}(T)} \end{aligned} \tag{4.8}$$

4.2.4 Synthèse

Nous avons vu dans cette partie comment modéliser des erreurs intermittentes à partir d'un modèle de Markov. Cela nous a permis de mettre en place une métrique : la probabilité de détection d'un test périodique (voir équation (4.4)). Cette métrique décrit la probabilité qu'une faute se déclare sur une durée d'observation mais soit détectée lors de n'importe lequel des tests exécutés durant cette durée. Elle dépend du type d'erreur à détecter (temps moyen de maintient de l'erreur et temps moyen entre deux erreurs), de la durée d'observation et de l'intervalle entre chaque tests effectués.

Afin de se rendre compte des contraintes liées à cette métrique, la partie suivante illustrera l'impact du type d'erreur et de la durée d'observation sur la probabilité de détection. Plus particulièrement nous illustrerons les résultats des équations (4.3) et (4.4) dans plusieurs cas d'étude.

4.3 Application à des cas d'étude

La partie précédente montre l'étude théorique qui nous a permis de calculer notre métrique : la probabilité de détection du test périodique. Dans cette partie, nous allons appliquer les formules précédentes à plusieurs cas d'étude. Cela permettra de mieux comprendre comment évolue la probabilité de détection, en fonction du temps et des différents types d'erreurs intermittentes.

4.3.1 Cas d'étude

Afin d'illustrer les équations de la partie précédente, nous allons définir plusieurs cas d'étude. Ces cas d'étude correspondent à différentes définitions d'erreurs intermittentes. Ainsi, chaque cas correspond à un couple λ et μ. Pour rappel, $1/\mu$ est la durée moyenne des erreurs et $1/\lambda$ le temps moyen sans erreur (voir § 4.2.2 page 81). Les unités de temps utilisées dans cette partie sont exprimées en millisecondes, ce choix est arbitraire mais permet de se rendre compte de l'impact des paramètres sur les probabilités. En effet, il n'existe pas suffisamment d'informations sur les erreurs intermittentes pour nous permettre de nous baser sur des données réelles pour définir nos cas d'études. Ainsi, nous avons définis des cas très différents permettant d'illustrer l'impact du type d'erreur et de la durée d'observation sur la probabilité d'occurrence des erreurs et la probabilité de détection.

Le tableau 4.1 montre les différents cas étudiés dans cette partie. Dans les cas 1 et 2, le système est respectivement mille fois et cent fois plus souvent dans l'état opérationnel que dans l'état défaillant. Le temps moyen avant l'apparition d'une erreur est respectivement de 1000 ms et 100 ms. Et dans les deux cas le temps moyen d'activation de l'erreur est de 1 ms.

Dans les cas 3 et 4, le système est la moitié du temps dans l'état opérationnel et la moitié du temps dans l'état défaillant.

Les cas 5 et 6 correspondent à une erreur permanente mais avec un temps moyen avant l'apparition d'une erreur différent.

Cas	μ/λ	λ (ms^{-1})	μ (ms^{-1})	$1/\lambda$ (ms)	$1/\mu$ (ms)
1	1000	0,001	1	1000	1
2	100	0,01	1	100	1
3	1	0,001	0,001	1000	1000
4	1	0,01	0,01	100	100
5	0	0,001	0	1000	∞
6	0	0,01	0	100	∞

TABLE 4.1 : *Définitions des cas d'étude.* $1/\mu$ *= durée moyenne des erreurs et* $1/\lambda$ *= temps moyen sans erreur.*

4.3.2 Probabilité d'occurrence d'une erreur intermittente

La figure 4.5 montre la probabilité $q(t)$ qu'une faute se déclare en fonction du temps, pour chacun des cas étudiés, et pour t variant de 0 à 2 s. Elle se base sur l'équation (4.3) page 83.

Cette figure permet de faire deux constats. Le premier constat concerne la probabilité que le système soit en erreur à $t = 0$. Cet instant, représente le début de l'observation des erreurs et correspond à une date quelconque dans la vie du système. Par conséquent, si un système est soumis à une erreur intermittente d'un des cas étudié, alors $q(0)$, cela correspond à la probabilité que le système soit en erreur dès le début de l'observation.

À $t = 0$, $q(0)$ est représenté par l'état stationnaire $\pi_1 = \frac{\lambda}{\lambda+\mu}$. Dans les cas 1 et 2, le rapport μ/λ est très grand donc la probabilité d'être dans un état défaillant à $t = 0$ est très faible.

Dans les cas 3 et 4, le système reste le même temps dans chacun des états, la probabilité d'être dans un état défaillant à $t = 0$ est donc de 50%.

Dans les cas 5 et 6, $\mu = 0$ donc à partir de $t = 0$ la probabilité d'être dans un état défaillant est maximale.

On peut remarquer que ces probabilités sont valables aussi bien pour les tests que pour les applications. Si l'on considère un système soumis à une erreur intermittente du cas 3 ou 4, alors au démarrage d'une application quelconque, celle-ci a une probabilité de 50% de s'exécuter alors que le système est défaillant.

Le deuxième constat concerne la durée d'observation. La figure 4.5 présente la probabilité $q(t)$ qu'une faute se déclare sur l'intervalle $[0;2s]$. Ainsi, dans ce cas, la durée maximale d'observation est de 2 s. Or, si nous prenons les cas 1 et 3, nous observons que la période moyenne de l'erreur $(1/\mu + 1/\lambda)$ est supérieure à 1 s. Cette valeur est trop proche de la durée maximale d'observation, ce qui explique la faiblesse des probabilités concernant ces deux cas. La probabilité qu'une faute se déclare sur une durée de 1 s dans le cas 1 est $(1 - e^{-\lambda}) = 63\%$.

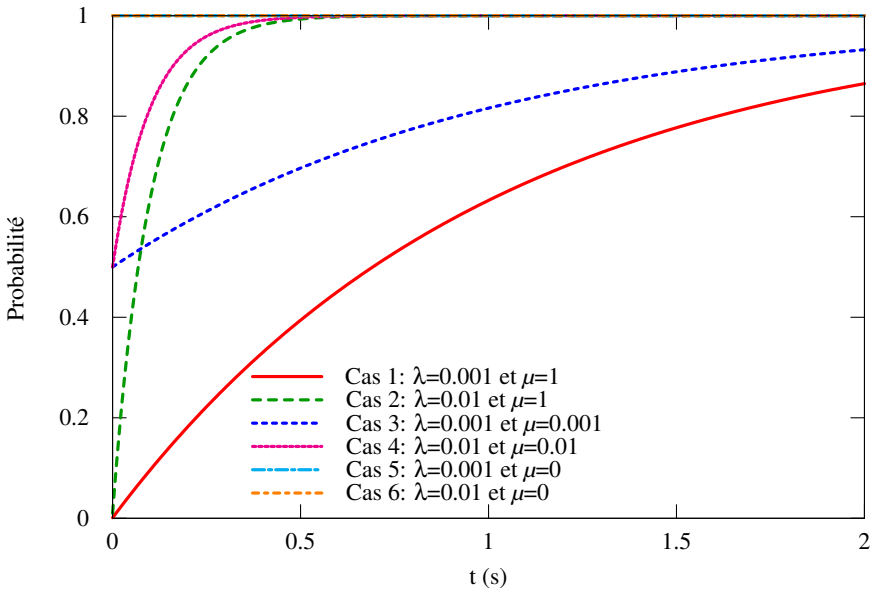


FIGURE 4.5 : *Probabilité qu'une faute se déclare pour chacun des cas d'étude*

4.3.3 Évolution de la probabilité de détection

La figure 4.6 présente la probabilité qu'une erreur intermittente s'active sur une durée de 1 s mais soit détectée pendant n'importe lequel des N tests. Cette probabilité correspond à l'équation (4.4) page 83. Elle exprime $d(N)$, N étant ici le nombre de tests réalisés sur 1 s. Pour simplifier, nous considérons ici un test périodique pour lequel tous les intervalles de test sont séparés d'une même durée $T = 1s/N$, la période de test.

Dans le cas 1, on constate que la probabilité de détection reste faible même avec un grand nombre de tests (40% pour 10^{10} tests sur 1 s). Cela est dû à une fenêtre d'observation trop courte, ce qui s'ajoute à la remarque du paragraphe précédent.

Dans le cas 2, la fenêtre d'observation est adaptée à $1/\lambda$ et on observe bien l'augmentation de la probabilité de détection en fonction du nombre de tests.

Dans les cas 3 et 4, le temps moyen dans l'état opérationnel est le même que celui dans l'état défaillant et on constate qu'une grande période de tests suffit à détecter une erreur. Seule la fenêtre d'observation a un impact sur la probabilité de détection. Dans le cas 3, il faudrait une fenêtre d'observation plus grande : la probabilité de détection passe à 93% en faisant cinq tests sur 5 s, ce qui équivaut à une période de test de 1 s.

Les cas 5 et 6 correspondent à des erreurs permanentes, un test suffit pour détecter l'erreur. La différence entre les deux cas est la même que pour les cas 3 et 4, il faudrait attendre 2.4 s pour avoir une probabilité supérieure à 90% dans le cas 4.

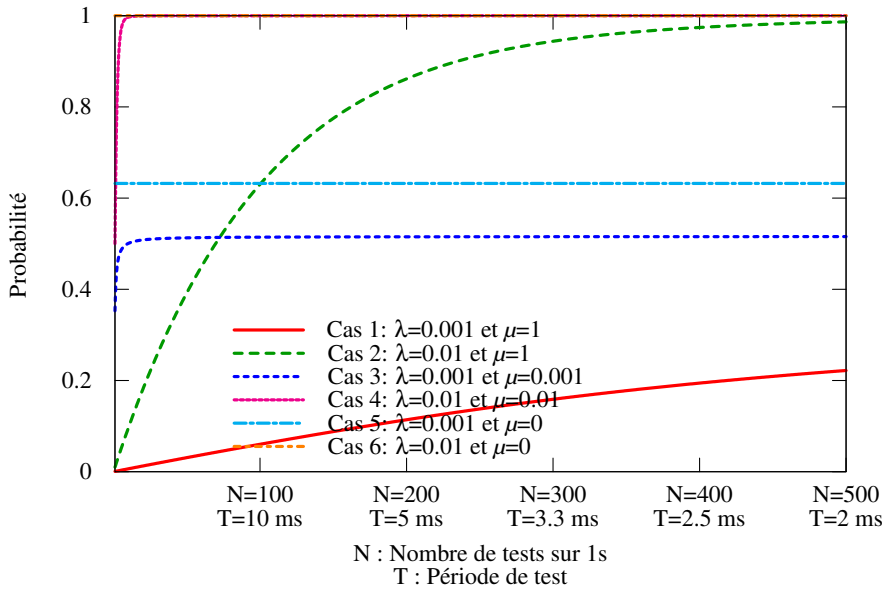


FIGURE 4.6 : *Probabilité qu'une faute se déclare sur l'intervalle* $[0, 1s]$ *et soit détectée lors d'un des* N *tests*

4.3.4 Synthèse

Cette partie nous a permis d'illustrer l'impact du type d'erreur et de la durée d'observation sur la probabilité d'occurrence des erreurs et la probabilité de détection.

En particulier, nous avons constaté qu'il est nécessaire d'avoir une bonne connaissance des erreurs à détecter, pour pouvoir mettre en place un test périodique. Un même test n'aura pas la même efficacité suivant le type d'erreur à détecter. De même, nous avons constaté qu'il est important d'adapter la fenêtre d'observation au type d'erreurs à détecter.

Dans cette partie, nous avons considéré un test périodique à période fixe dans le but de simplifier nos observations. Cependant, le but de ce chapitre est d'analyser l'efficacité d'un test pseudo-périodique sur la probabilité de détection, c'est l'objectif du prochain chapitre.

4.4 Étude des tests pseudo-périodiques

Cette partie débute par la définition des tests pseudo-périodiques. En particulier, cette définition permettra de différencier ce type de test avec un test périodique à période fixe.

Nous définirons ensuite un pire cas de test pseudo-périodique. Cela nous permettra de borner l'impact des tests pseudo-périodiques sur la probabilité de détection. Nous confirmerons ainsi l'efficacité de ces tests.

4.4.1 Définition du test pseudo-périodique

Nous définissons un test pseudo-périodique par un test périodique dont l'intervalle entre chaque test n'est pas constant. À l'inverse un test périodique à période fixe est un test périodique pour lequel l'intervalle entre chaque test est strictement égale à la période de test.

Or, comme nous l'avons vu dans la partie 4.2.3, la probabilité de détection d'un test est défini en fonction de l'intervalle entre chaque test. Ainsi, nous voulons observer l'impact d'une distribution non uniforme des intervalles de test sur la probabilité de détection.

Pour cela nous allons considérer l'intervalle entre deux tests comme une variable aléatoire. Avec une moyenne égale à la période de test T et un écart type égal à $v.T$ avec v compris entre 0 et 1 (voir figure 4.7). Ainsi, l'écart type est une fraction de la période de test. Si $v = 0$ alors il n'y a aucune variation et le test est strictement périodique de période T. Si $v = 1$, alors cela revient à effectuer seulement un test sur deux. Dans ce cas, le test est strictement périodique, mais de période$2.T$.

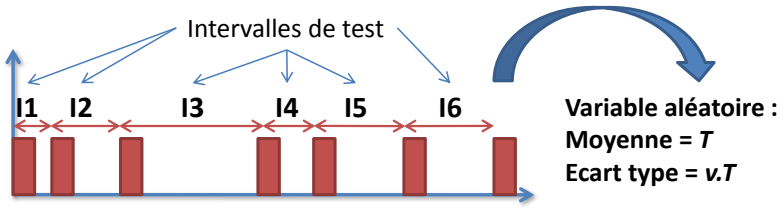


FIGURE 4.7 : *Définition du test pseudo-périodique. On considère l'intervalle entre deux tests comme une variable aléatoire, ayant une moyenne égale à la période de test T et un écart type égal à $v.T$ avec v compris entre* 0 *et* 1.

4.4.2 Pire cas de variation des intervalles de test

Après une étude statistique de l'impact de la distribution des intervalles de test sur la probabilité de détection nous avons déduit un pire cas. Ce pire cas correspond à une distribution spécifique des intervalles de test ayant pour moyenne T et écart type $v.T$. Cela nous permet de borner la probabilité de détection pour un v et un T donnés.

Si on considère toutes les configurations de distribution des test possibles, nous définissons un pire cas pour lequel la probabilité de détection est la plus faible. Ce cas est défini par une alternance des intervalles de test entre $(1-v).T$ et $(1+v).T$. La figure 4.8 illustre la répartition statistique dans ce pire cas et le compare avec le cas idéal d'un test strictement périodique. Dans les deux cas la population est de 10. Dans le premier cas, la moitié des intervalles de test est séparés de $(1-v).T$ et l'autre moitié de $(1+v).T$. Dans le deuxième cas, tous les intervalles de test sont séparés de la période de test T.

La figure 4.9 illustre le cas idéal où $v = 0$, et la configuration du pire cas pour deux valeurs de v. Dans le premier cas, $v = 1$ ce qui équivaut à un test périodique de période $2T$. Dans le deuxième cas, $v = 0,5$ donc la moitié des intervalles de test est séparés d'une durée $(1-v).T = 0,5.T$ et l'autre par une durée $(1+v).T = 1,5.T$.

À l'aide de la formule (4.4), nous pouvons définir la probabilité de détection dans la configuration du pire cas par :

$$\begin{array}{l} \forall v \in [0;1] \\ d_{wc}(N) \quad = \quad q(N.T).(1-P_{0,0}((1-v).T).P_{0,0}((1+v).T))^{N/2} \end{array} \tag{4.9}$$

Et on a :

$$d_{wc}(N) \quad < \quad d(N) \tag{4.10}$$

L'équation (4.9) permet de borner la probabilité de détection d'un test pseudo-périodique ayant une distribution des intervalles de test de moyenne T et écart type $v.T$.

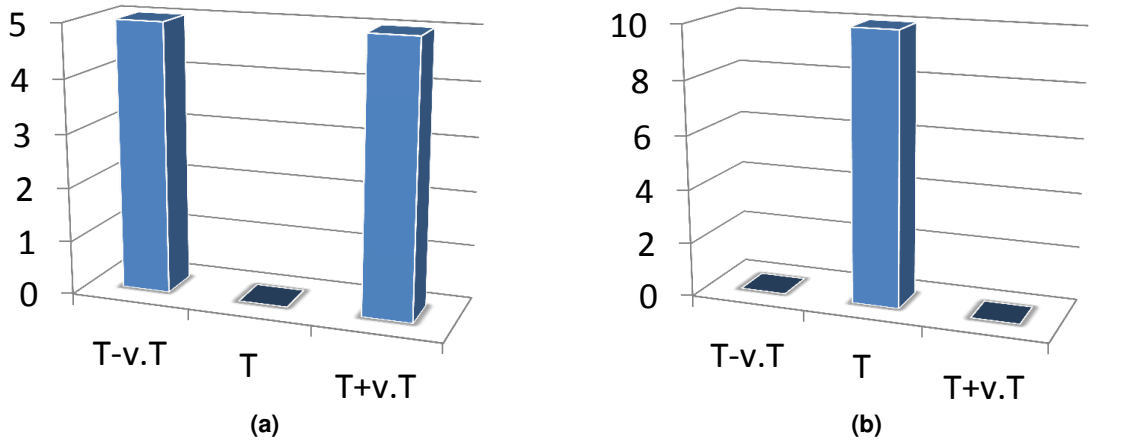


FIGURE 4.8 : *Répartition statistique des intervalles de test pour une population de 10. (a) Dans le pire cas, la moitié des intervalles de test sont séparés de $(1-v).T$ et l'autre moitié de $(1+v).T$. (b) Dans le cas idéal tous les intervalles de test sont séparés de la période de test T, le test est strictement périodique.*

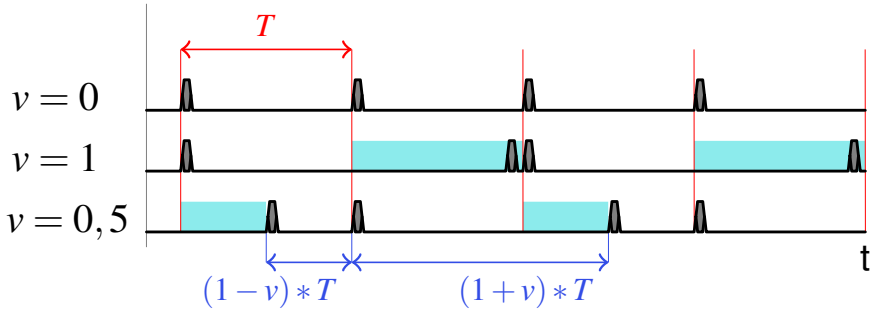


FIGURE 4.9 : *Illustration de la variation des intervalles de test. Exemples de pires cas pour une distribution statistique des intervalles de test, avec pour moyenne T (la période de test), et écart type $v.T$. Le cas $v = 0$ est le cas idéal.*

4.4.3 Impact de la variation des intervalles de test sur la probabilité de détection

Ce paragraphe a pour but d'étudier l'impact d'un test pseudo-périodique sur la probabilité de détection. Tout d'abord, nous présentons les périodes de test obtenues pour chacun des cas d'études présentés dans la partie précédente. Puis, nous présenterons les résultats d'analyse à l'aide de probabilités de détection du test pseudo-périodique dans le pire cas.

Calcul des périodes de test

Le tableau 4.2 présente les périodes de test (T) calculées pour chacun des cas d'étude, et pour obtenir une probabilité de détection de $0,999$ après une durée W, dans le cas d'un test d'une période fixe ($v = 0$). W représente la taille de la fenêtre d'observation et a été fixée à $W = 10/\lambda$. Cela résulte des observations des paragraphes précédents.

Nous pouvons constater que la période de test dans les cas 1 et 2 est très faible. Dans ces deux cas, la durée moyenne de l'erreur ($1/\mu$) est de 1 ms, ainsi, la période de test doit être dans le même ordre de grandeur.

Les cas 5 et 6 représentent des erreurs permanentes, ainsi, seul un test est nécessaire à la fin de la fenêtre d'observation pour déterminer s'il y a une erreur.

Cas	μ/λ	λ (ms^{-1})	μ (ms^{-1})	T (ms)	N	W (ms)
1	1000	0,001	1	0,776	12881	10000
2	100	0,01	1	0,775	1290	1000
3	1	0,001	0,001	655,738	15	10000
4	1	0,01	0,01	65,574	15	1000
5	0	0,001	0	9900	1	10000
6	0	0,01	0	990	1	1000

TABLE 4.2 : *Périodes de test en fonction des cas. Résultats obtenus pour atteindre une probabilité de détection de $0,999$ au bout d'une durée W (taille de la fenêtre d'observation), soit après N tests.*

Probabilité de détection d'un test pseudo-périodique

Pour étudier l'impact de la variation des intervalles de test sur la probabilité de détection, nous allons utiliser l'équation (4.9) page 90. Cette équation définie d_{wc}, la probabilité de détection minimale d'un test pseudo-périodique, ayant une distribution statistique des intervalles de test de moyenne T et d'écart type $v.T$.

La figure 4.10 montre l'impact de la variation des intervalles de test sur la probabilité de détection. Seuls les cas 2, 4 et 6 sont présentés. En effet, pour comparer les cas, ils doivent avoir

une durée d'observation identique (ici $W = 1000ms$). La période de test a été calculée de manière à obtenir une probabilité de détection de $0,999$ au bout de 1000 ms.

Nous pouvons constater que l'impact de la variation des intervalles de test est différent suivant les cas étudiés. Concernant le cas 6, l'erreur étant permanente la probabilité de la détecter est maximale quelles que soient les variations. Concernant les cas 2 et 4, la différence est visible pour des variations supérieures à 30% de la période de test initiale.

Si l'on prend le cas 4 représenté sur la figure, nous pouvons calculer que si $v = 40\%$ alors $d_{wc} = 0.9985$ ce qui représente un écart de moins de $0,04\%$ sur la probabilité de détection visée. Ce résultat peut être utilisé de deux façons.

Premièrement, si l'on connait la variation induite par l'implémentation du test sur un système donné, alors la période de test peut être ré-évaluée pour prendre en compte les variations. Dans notre cas, la période de test devrait être réduite de $12,2\%$. C'est à dire qu'elle passerait de $T = 65,57$ ms à $T = 57,57$ ms.

Deuxièmement, si l'on a $0,04\%$ de tolérance sur la probabilité de détection, alors le test peut tolérer des variations des intervalles de test jusqu'à 40% de la période de test.

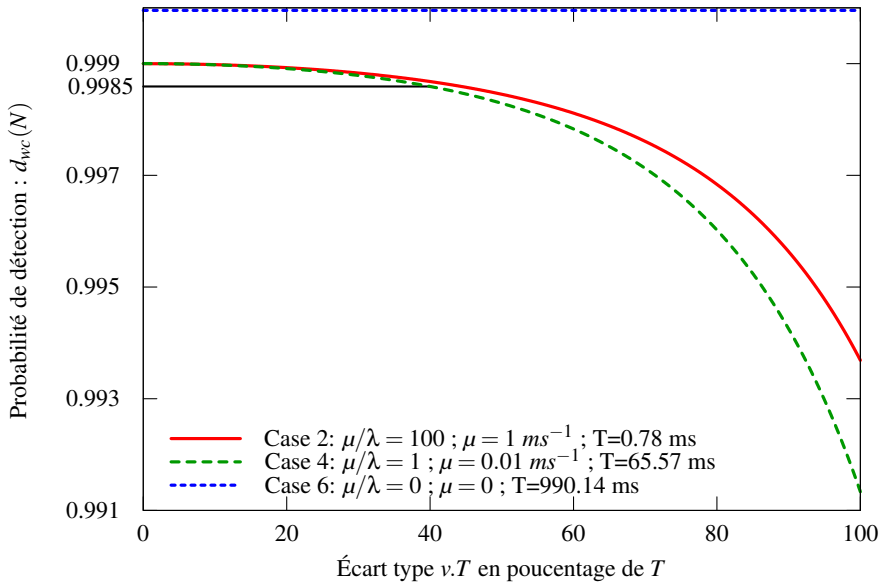


FIGURE 4.10 : *Impact de la variation des intervalles de test sur la probabilité de détection (d_{wc}) pour une probabilité de détection de $0,999$ après une durée $W = 1000$ ms.*

Test Pseudo-Périodique

L'objectif de cette partie était d'analyser l'impact de la variation des intervalles de test sur la probabilité de détection. En effet, nous envisageons que le test ne soit pas prioritaire devant les applications. Cela a un impact sur la périodicité du test. Même si la période est respectée en moyenne, des variations peuvent apparaitre rendant le test pseudo-périodique.

Nous avons montré, sur un exemple, que si une erreur de $0,04\%$ est tolérée sur la probabilité de détection alors le système peut tolérer un test pseudo-périodique ayant des variations jusqu'à 40% de la période de test.

Ce résultat confirme que le test ne doit pas nécessairement être prioritaire devant les applications. Cela permettra de réduire les contraintes du test sur l'ordonnancement des applications.

Dans cette partie, nous avons considérés des cas d'étude théoriques. Or, l'objectif est que ces tests soient implémentés matériellement. Dans ce sens, la partie suivante explique comment configurer un test périodique à partir des informations de ce chapitre.

4.5 Configuration du test

Cette partie explique comment déterminer la période du test périodique. Elle insiste plus particulièrement sur le choix des erreurs à détecter et l'adéquation du test avec les applications.

4.5.1 Détermination de la période de test

Pour déterminer la période de test nécessaire à la détection des erreurs, il est nécessaire d'avoir une définition de ces dernières. Ensuite, les équations définies précédemment permettent de calculer la période de test nécessaire pour atteindre une probabilité de détection donnée après un temps donné. La figure 4.11 illustre ce procédé.

Voici un exemple :

Définition de l'erreur : Comme nous l'avons vu au cours de ce chapitre, la définition du type d'erreur est à la base du test périodique et défini son efficacité. Une erreur intermittente est définie par $1/\mu$, la durée moyenne des erreurs et par $1/\lambda$, le temps moyen entre deux erreurs.

Cette définition peut être obtenue en ayant une connaissance du matériel. Par exemple, une expérience comme celle présentée dans le chapitre 2 permet d'obtenir cette définition dans des conditions données. Cependant, la définition des erreurs intermittentes peut aussi être donnée par les applications (voir paragraphe suivant). Dans ce cas, elle représente une limite de tolérance de l'application.

Pour exemple prenons $\lambda = 0,01\ \text{ms}^{-1}$ et $\mu = 0,1\ \text{ms}^{-1}$. Cela correspond à une durée moyenne entre deux erreurs de 100 ms et à une durée moyenne d'erreurs de 10 ms. Dans ce cas, le système est dix fois plus souvent dans l'état opérationnel que dans l'état défaillant.

Probabilité de détection : La probabilité de détection permet de définir la précision du test, mais aussi le coût de celui-ci. En effet, plus la probabilité de détection à obtenir est élevée et plus la période de test devra être faible. Pour l'exemple, prenons une probabilité de détection de trois neufs, soit $0,999$. Cette valeur représente une probabilité relativement importante et sera utilisée comme valeur de référence dans la suite de ce mémoire.

Durée du test : Comme nous l'avons montré dans ce chapitre, la durée d'observation doit être réfléchie en fonction de la définition de l'erreur à détecter. La durée de test définit le temps

requis pour atteindre la probabilité de détection visée. Pour l'exemple, prenons une durée de 1 s qui équivaut à $10/\lambda$.

Modèle de Markov : Le modèle utilisé défini le lien entre les paramètres précédents et la période de test. Pour cela nous avons défini plusieurs équations permettant de définir la probabilité de détection d'un test périodique. Par exemple, nous pouvons nous baser sur l'équation (4.7) si le test est strictement périodique ou sur l'équation (4.9) si le test est pseudo-périodique et que l'on connait la distribution des intervalles de test.

Période de détection : En prenant en compte les paramètres précédents, nous pouvons déterminer la période de test nécessaire pour obtenir une probabilité de détection de $0,999$ après 1 s.

Dans le cas d'un test périodique à période fixe la période sera de $7,63$ ms, ainsi, il sera nécessaire de réaliser 131 tests en 1 s.

Dans le cas d'un test pseudo-périodique dont la distribution des intervalles de test à un écart type de 40%, alors la période de test nécessaire est de $6,74$ ms, soit 148 tests en 1 s.

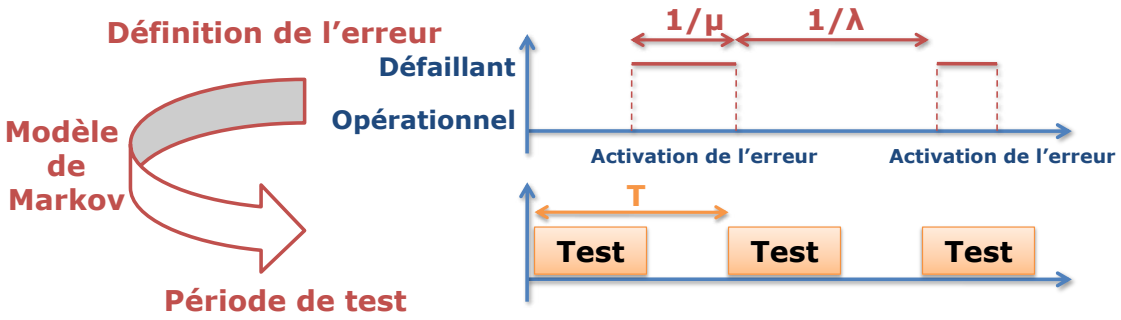


FIGURE 4.11 : *Illustration du calcul de la période de test. A partir d'une définition de l'erreur intermittente à détecter (durée moyenne des erreurs et durée moyenne sans erreur) et du modèle de Markov, il est possible de déterminer la période de test nécessaire pour atteindre une probabilité de détection donnée sur un temps donné.*

4.5.2 Adéquation du test avec les applications

Le but du test périodique est de détecter les erreurs intermittentes pour éviter que les applications ne produisent des résultats erronés. Cependant, les applications peuvent tolérer naturellement des erreurs. Par exemple, si l'on considère une application de traitement d'images, nous pouvons imaginer que la perte d'une partie de l'image peut être tolérée si elle n'est pas trop fréquente. Dans ce sens, la définition de l'erreur à détecter peut être donnée par les applications. Ainsi, le développeur des applications est le mieux placé pour définir le type d'erreur à détecter.

En pratique : Considérons qu'une application de traitement d'images peut tolérer deux images (ou parties de l'image) consécutives erronées toutes les cent images et que le traitement est réalisé à 26 images par secondes. Nous pouvons alors calculer la définition de l'erreur à détecter. Celle-ci est définie à partir de deux paramètres : la durée moyenne des erreurs et la durée moyenne sans

erreur. Ainsi, ces deux paramètres sont respectivement égaux à des durée de 2 et 98 images, soit : $1/\mu = 77\ ms$ et $1/\lambda = 3,769\ s$.

Nous verrons dans le chapitre suivant comment l'implémentation d'un test pseudo-périodique a un impact sur les applications. Cela confirmera que l'implémentation d'un test périodique doit être réfléchie en adéquation avec les applications.

4.6 Conclusion

L'objectif de ce chapitre était d'étudier la faisabilité d'utiliser un test pseudo-périodique pour détecter les erreurs intermittentes. En effet, contrairement aux approches utilisées dans la littérature, nous pensons que le test en ligne ne doit pas nécessairement être prioritaire et donc strictement périodique. Pour répondre à cet objectif, trois parties principales se sont succédées, une partie théorique, une partie appliquée à des cas d'étude et une partie dédiée à l'étude des tests pseudo-périodiques.

Nous avons montré dans la première partie qu'il est possible de modéliser les erreurs intermittentes à l'aide d'un modèle de Markov. Cela nous a permis de décrire la probabilité de détection en fonction des caractéristiques des erreurs à détecter et de la période de test.

La deuxième partie de ce mémoire, nous a permis d'étudier l'impact de différents cas d'études sur la probabilité d'observer et de détecter une erreur en fonction du temps. Nous avons, en particulier, démontré qu'il est important de connaitre le type d'erreur à détecter afin d'adapter au mieux la durée d'observation des erreurs.

Finalement, nous avons conclu dans une troisième partie, que l'utilisation d'un test pseudo-périodique est possible. Si une erreur de $0,04\%$ est tolérée sur une probabilité de détection de $0,999\%$, alors le système peut tolérer des variations jusqu'à 40% de la période de test initiale.

Pour compléter cette étude théorique, nous devons implémenter notre méthode de test pseudo-périodique dans une architecture multiprocesseur. En effet, l'étude présentée dans ce chapitre ne considère qu'un processeur et ne tient pas compte de conditions réelles de fonctionnement d'une architecture complète. Ainsi, le principal objectif du prochain chapitre, sera de mettre en œuvre et d'évaluer plusieurs implémentations de tests périodiques et pseudo-périodiques, dans différentes conditions de fonctionnement (différents taux d'occupation de l'architecture et différents types d'applications). En particulier, nous utiliserons la probabilité de détection définie dans ce chapitre, comme une métrique permettant d'évaluer l'efficacité de la détection des erreurs intermittentes.

CHAPITRE 5

Implémentation d'un test pseudo-périodique sur une architecture multiprocesseur

Sommaire

LES CHAPITRES précédents nous ont permis d'identifier une méthode de test capable de détecter des erreurs intermittentes dans une architecture multiprocesseur. Tout d'abord, nous avons sélectionné les méthodes de Software-Based Selt-Test (SBST) ou des méthodes basées sur des structures de BIST, comme moyen de détecter des erreurs intermittentes dans des processeurs. En effet, combinées à un test périodique, ces méthodes offrent un bon compromis entre la couverture de fautes et l'impact sur les performances. De plus, elles permettent d'être implémentées dans une architecture multiprocesseur, et ne nécessitent pas de surface additionnelle.

Dans le but de réduire l'impact du test au minimum, nous avons étudié l'impact d'un test périodique non prioritaire sur la probabilité de détection du test. Ainsi, nous avons montré qu'un test pseudo-périodique reste efficace même avec de fortes variations des intervalles de test. Théoriquement, nous savons que les tests pseudo-périodiques peuvent détecter des erreurs intermittentes, cependant nous devons déterminer leur impact sur les performances des applications dans des cas réels d'exécutions. Dans ce sens, ce chapitre détaillera l'implémentation du test en ligne dans une architecture matérielle et démontrera, à partir de simulations de l'architecture complète, que l'utilisation de tests pseudo-périodiques est adaptée aux architectures multiprocesseur embarquées.

Pour arriver à cette conclusion, nous présenterons nos objectifs dans une première partie. Puis, nous décrirons dans une deuxième partie, l'environnement matériel et logiciel utilisé pour notre étude. C'est à dire, l'architecture multiprocesseur et les applications utilisées pour les simulations.

Dans la troisième partie de ce mémoire, nous expliquerons comment les tests pseudo-périodiques peuvent être intégrés dans une architecture multiprocesseur. Puis, nous détaillerons les différentes implémentations de tests pseudo-périodiques et strictement périodiques qui seront comparés par la suite.

La quatrième partie de ce chapitre présentera les différents paramètres utilisés dans les simulations. En particulier, nous présenterons les différentes configurations d'une application et de l'architecture, qui permettront d'étudier l'impact des tests pseudo-périodiques sur un large panel de cas d'études. Nous ferons ainsi varier les conditions expérimentales avec la charge de l'architecture et la nature des applications.

La dernière partie de ce chapitre, comparera l'impact des différentes implémentations des tests sur les applications et sur la probabilité de détection du test. Cela nous permettra de conclure sur l'efficacité des tests pseudo-périodiques. En particulier, nous montrerons qu'une politique pseudo-périodique prenant en compte les processeurs au repos et la priorité des tâches, offre le meilleur compromis entre performance et probabilité de détection.

5.1 Objectifs de l'étude

Dans une architecture multiprocesseur, plusieurs processeurs sont utilisés pour effectuer les traitements définis par une ou plusieurs applications. En parallèle, tous ces processeurs doivent être testés afin d'y détecter d'éventuelles erreurs intermittentes. Pour cela, nous avons vu précédemment qu'un test pseudo-périodique peut être utilisé. Cette technique de détection consiste à exécuter des tests avec une période qui dépend du type d'erreur à détecter.

Le premier objectif de ce chapitre est de présenter comment les différentes tâches, des applications et des tests, doivent être distribuées sur les processeurs au cours du temps. Pour cela, l'architecture doit réaliser deux ordonnancements de manière concurrente :

L'ordonnancement des applications : il définit la manière dont les tâches applicatives sont distribuées sur les processeurs. Cela est réalisé par un algorithme d'ordonnancement dont le but est de minimiser le temps d'exécution des applications. Cet algorithme n'est pas le sujet d'étude de ce chapitre, donc nous ne le modifierons pas.

L'ordonnancement des tests : il définit l'ordre dans lequel les processeurs doivent être testés. Cet ordonnancement est le sujet de ce chapitre. Nous allons étudier plusieurs politiques d'ordonnancement des tests implémentant des tests pseudo-périodiques. Le but est de sélectionner l'ordonnancement qui a le plus faible impact sur les applications (durée, nombre de préemptions, etc), et qui permet d'atteindre une probabilité de détection fixée sur chacun des processeurs.

Le deuxième objectif est de confirmer dans des conditions réelles de fonctionnement, que l'utilisation de tests pseudo-périodiques est préférable à l'utilisation de tests strictement périodiques. En particulier, qu'ils minimisent l'impact du test sur les performances des applications et qu'ils conservent la probabilité de détection.

Pour atteindre nos objectifs nous utiliserons une architecture spécifique et une application dédiée. L'architecture nous permettra d'implémenter plusieurs politiques de tests pseudo-périodiques et l'application nous permettra de faire varier les conditions de simulation. Dans ce sens, la prochaine partie présente l'architecture multiprocesseur SCMP [75] et son simulateur, ainsi que l'application de Labelling[76].

5.2 Environnement de l'étude

Cette partie présente l'environnement matériel et logiciel utilisé pour étudier l'implémentation des tests pseudo-périodiques dans une architecture multi-processeur. En particulier, nous présentons l'architecture SCMP [75] et son simulateur SESAM [77], ainsi que l'application de Labelling [76].

Cette partie doit permettre d'avoir une idée générale des outils utilisés pour réaliser notre étude. Elle aide à mieux comprendre quels éléments ont été modifiés pour intégrer la détection des erreurs intermittentes au sein de l'architecture. Ce sera l'objet de la prochaine partie.

5.2.1 L'architecture multiprocesseur

L'architecture multiprocesseur utilisée dans notre étude est SCMP (SCalable Multi-Processor) [75]. Comme le montre la figure 5.1, elle est considérée comme un accélérateur matériel et pour fonctionner elle doit être couplée à un processeur. Un système d'exploitation est exécuté sur le processeur, c'est lui qui envoie les demandes d'exécution à SCMP. SCMP est dédié au calcul intensif et bénéficie de mécanismes optimisés pour la gestion des préemptions et la migration des tâches. La communication entre les deux éléments est effectuée par des interruptions et par une mémoire partagée.

Plus précisément, l'architecture utilisée pour notre étude est composée d'une partie contrôle (l'OSoC), de processeurs, de mémoires et d'un réseau interconnexion.

La partie suivante détaille l'architecture SCMP et son modèle d'exécution.

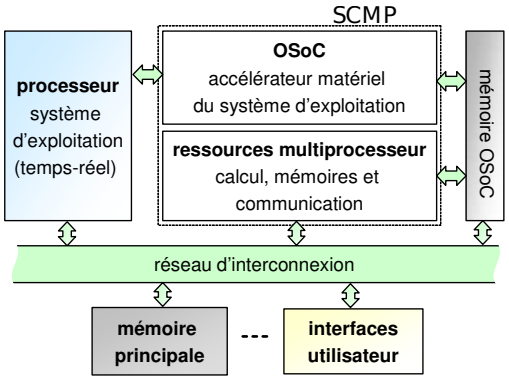


FIGURE 5.1 : *Intégration de l'architecture SCMP dans un système global [75]. SCMP est un accélérateur matériel permettant de réaliser du calcul intensif.*

Détail de l'architecture SCMP

La figure 5.2 détaille la structure de l'architecture SCMP. L'élément principal de l'architecture est l'OSoC (Operating System accelerator on Chip). Il est chargé de contrôler dynamiquement l'architecture SCMP, c'est lui qui ordonnance l'exécution des applications sur les différentes ressources de calcul disponibles. La modification de ce composant nous permettra d'intégrer notre technique de test pseudo-périodique au sein de l'architecture.

La communication entre les ressources de calcul et le contrôleur est assurée par l'Unité d'Interface et d'Interconnexion avec l'OSoC. Pour cela, chaque ressource de calcul comporte une Interface avec l'OSoC (IO). L'Unité de Gestion de la Mémoire (UGM) permet aux Générateurs d'Adresse (GA) d'accéder directement aux instructions et aux données de la tâche en mémoire.

L'architecture SCMP est basée sur des mémoires physiquement distribuées et logiquement partagées. Les éléments de calculs sont connectés aux processeurs par un réseau multibus. Chaque processeur a un accès en lecture à toutes les mémoires et un accès privilégié en écriture à une zone mémoire dédiée.

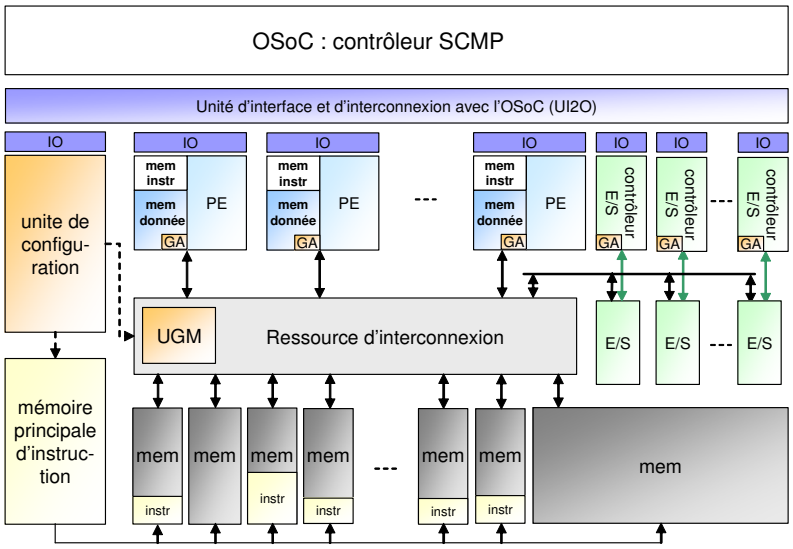


FIGURE 5.2 : *Description de l'architecture SCMP [75]. UI2O : Unité d'Interface et d'Interconnexion de l'OSoC, réalise l'interconnexion entre les ressources calcul et l'OSoC. IO : Interface avec l'OSoC, interface entre les ressources de calcul et l'UI2O. GA : Générateurs d'Adresse (GA). UGM : Unité de Gestion de la Mémoire, permet aux GA d'accéder aux instructions et aux données de la tâche en mémoire.*

Modèle d'exécution

Pour être utilisées par l'OSoC, les applications doivent être découpées en tâches indépendantes. La dépendance de contrôle et de données entre les tâches est décrite par un réseau de Pétri [78]. Le code des tâches et le graphe de l'application est transmis à l'OSoC par le système d'exploitation par l'intermédiaire de la mémoire partagée (voir figure 5.1). Nous présenterons dans la partie 5.2.3 l'application utilisée pour nos simulations.

Après la réception d'une demande d'exécution, l'OSoC configure les tâches à exécuter. Pour cela, pour chaque tâche, les instructions sont préchargées dans une des mémoires partagées. L'adresse de la mémoire dédiée à la tâche est sauvegardée par l'UGM. Dès qu'une tâche peut être exécutée, elle est allouée sur une ressource de calcul disponible. Celui-ci récupère l'adresse de la mémoire dédiée à la tâche par l'UGM et accède aux informations de la tâche par son GA. La tâche est ensuite exécutée sur le processeur et des données sont produites dans une ou plusieurs mémoires. Ces données deviennent disponibles pour les tâches suivantes.

En fonction de l'algorithme d'ordonnancement implémenté, une tâche en cours d'exécution peut être préemptée. Dans ce cas, le contexte d'exécution de la tâche et les données intermédiaires sont sauvegardés dans la mémoire dédiée à la tâche. Ainsi, la même tâche peut reprendre son exécution sur un autre processeur avec une faible pénalité. En effet, toutes les mémoires sont partagées et la latence d'accès aux mémoires est la même pour tous les processeurs. Le temps de préemption est constant et comprend la sauvegarde de la mémoire interne du processeur et la latence du réseau d'interconnexion.

Dans notre cas, l'algorithme d'ordonnancement est ELLF (Enhanced Least-Laxity-First) [79]. Pour cet algorithme, la tâche dont la laxité est la plus faible est la tâche la plus prioritaire. Cet

algorithme est exécuté avec une période fixe. Ainsi, l'ordonnancement des tâches est remis en jeu à chaque nouvelle période. Le choix de cette période est important, trop faible, elle nuit aux performances et trop importante elle induit une sur-consommation.

Le paragraphe suivant présente le simulateur utilisé pour obtenir nos résultats.

5.2.2 Le simulateur

Le simulateur SESAM [77] (Simulation Environment for Scalable Asymmetric Multiprocessors) utilisé pour notre étude, a été conçu dans le but de réaliser l'exploration d'architectures multiprocesseur. C'est un environnement modulaire utilisant des ISS (Instruction Set Simulator) pour modéliser les processeurs et SystemC [80] pour le reste de l'architecture. Il permet de modifier facilement tous les paramètres de l'architecture, comme par exemple, le nombre de processeurs de l'architecture, les tailles mémoires, la latence d'accès aux mémoires ou les applications. Ainsi, il est un bon candidat pour évaluer nos politiques d'ordonnancement des tests.

L'architecture simulée se base sur l'architecture SCMP. Les processeurs sont de type MIPS32 et sont simulés par ArchC [81]. Ils sont connectés aux mémoires par un multibus, c'est à dire que chaque processeur accède à chaque mémoire avec la même latence. Le contrôle de l'architecture est réalisé matériellement par l'OSoC. Les différentes politiques d'ordonnancement des tests sont implémentées par ce composant.

Le paragraphe suivant décrit les applications utilisées dans nos simulations.

5.2.3 Les applications

Afin de simuler l'ajout d'un test pseudo-périodique en parallèle des applications, deux applications vont être utilisées. La première est l'application de labelling qui sera notre application principale. La deuxième représentera le test des processeurs.

L'application de labelling

L'application principale utilisée lors des simulations est basée sur une application de traitement d'image : l'application de labelling [76]. L'implémentation de ce type d'application dans des architectures multiprocesseur est souvent réalisée sur le même principe. C'est à dire que l'image à traiter est découpée en plusieurs parties qui sont traitées en parallèle par des processeurs différents. Ainsi, chaque processeur ne traite qu'une partie de l'image, les différentes parties sont assemblées dans une deuxième partie de l'application.

La figure 5.3 présente le graphe de tâches de l'application utilisée. La tâche T0 initialise l'application et les tâches de T1 à T8 exécutent les traitements de labelling. Dans notre cas, ces tâches sont exécutées N fois. Nous verrons dans la suite de ce chapitre, comment la durée de ces tâches influence la charge de l'architecture. Les tâches suivantes réalisent la fusion des résultats et clôturent l'application. Cependant, dans notre cas, la durée de ces tâches est minimisée en

utilisant des images ne nécessitant pas de fusion. Ainsi, ce sont les tâches T1 à T8 qui définiront la durée totale de l'application.

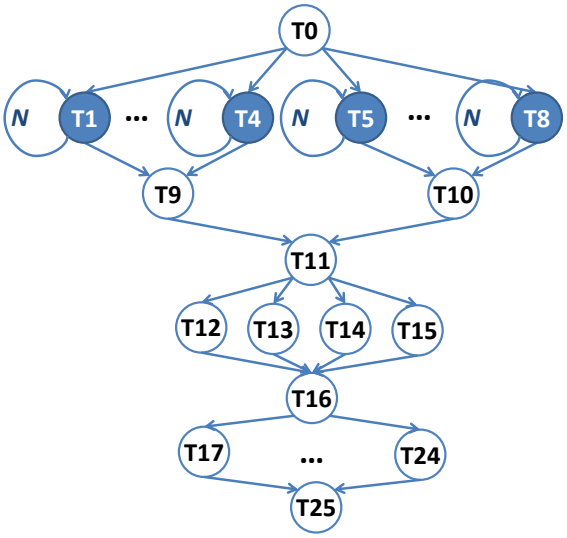


FIGURE 5.3 : *Application de labelling pour huit processeurs. La tâche T0 initialise l'application et les tâches de T1 à T8 exécutent les traitements de labelling. Dans notre cas, ces tâches sont exécutées N fois et sont les tâches principales. Les tâches suivantes réalisent la fusion des résultats.*

Implémentation du test pseudo-périodique

Le test pseudo-périodique est simulé par une tâche périodique dont la durée correspond à la durée des tests et la période est en rapport avec la période des tests. Nous verrons dans la partie suivante comment l'ordonnancement de l'OSoC a été modifié pour prendre en compte des tâches de test. Ainsi, la différence avec l'application précédente est que cette tâche est ordonnancée de façon spécifique en fonction d'une politique d'ordonnancement des tests choisie.

5.3 Intégration des tests

La partie précédente nous a permis de décrire l'architecture utilisée pour notre étude et l'application utilisée pour les simulations. Plus particulièrement, nous avons vu que l'ordonnancement des applications au sein de l'architecture multiprocesseur est effectué par l'OSoC. Cet ordonnanceur matériel nécessite d'être modifié pour implémenter des techniques de test pseudo-périodique. Nous présenterons, dans une première sous-partie, comment l'ordonnancement doit être modifié. Puis, dans une seconde sous-partie, nous détaillerons les différentes politiques d'ordonnancement des tests.

5.3.1 Ordonnancement et placement des applications

Comme nous l'avons vu dans la partie précédente, l'OSoC est chargé de l'ordonnancement et du placement des tâches au sein de l'architecture SCMP. Ici, nous allons nous concentrer sur ces deux étapes (voir figure 5.4).

L'ordonnancement des tâches dans une architecture est réalisé en ligne par un algorithme d'ordonnancement [82, 83], dans notre cas, l'algorithme utilisé est l'ELLF [79]. Son rôle est de répartir les tâches d'application sur tous les processeurs de l'architecture en fonction de leurs priorités. Dans le cas de l'algorithme ELLF, la tâche dont la laxité est la plus faible est la tâche la plus prioritaire. Les tâches sélectionnées par l'ordonnanceur, sont ensuite placées sur les différents processeurs.

Pour ajouter le test des processeurs en parallèle des applications, ces étapes doivent être modifiées. La suite de cette partie explique comment.

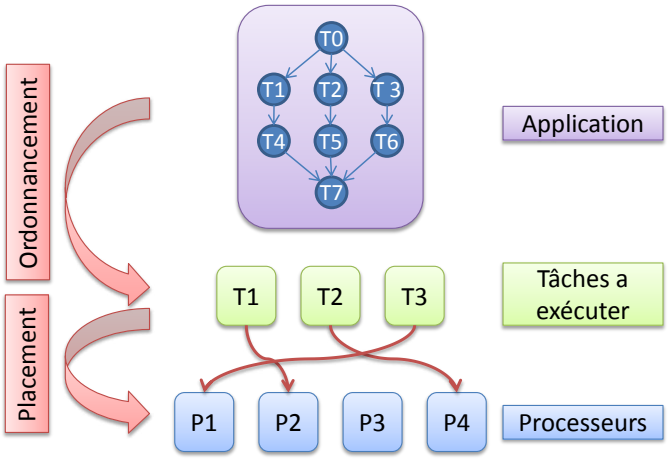


FIGURE 5.4 : *Ordonnancement et placement des tâches. L'algorithme d'ordonnancement sélectionne les tâches à exécuter en fonction de leurs priorités. Ces tâches sont ensuite placées sur les processeurs.*

5.3.2 Modification de l'ordonnancement pour les tests

Nous considérons ici un test périodique des processeurs. Soit T_{test} [1] la période de test d'un processeur et n le nombre de processeurs à tester, ainsi, un nouveau processeur sera testé tous les T_{test}/n. Cette période est celle de la tâche périodique implémentée dans l'OSoC qui représente le test des processeurs. Cette période est donc fixe pour l'ordonnanceur, cependant, nous verrons par la suite comment le choix des processeurs à tester induit un test pseudo-périodique.

Terminologie utilisée : Chaque processeur peut avoir un statut "testé" ou "non testé", si tous les processeurs sont "testés" alors le test est "complet". Le test des processeurs est alors "réini-

1. Le chapitre 4 explique comment choisir cette période.

tialisé" et tous les processeurs prennent le statut "non testé". Un processeur "doit être testé" s'il est "non testé" mais a été sélectionné pour être testé au cours de l'ordonnancement des tests.

De même, on considère que les processeurs peuvent avoir deux états : "occupé" ou "libre", suivant qu'une application est en cours d'exécution ou non. Si une application est en cours d'exécution, le processeur a une priorité définie par cette application. Cette priorité dépend de l'algorithme d'ordonnancement utilisé.

La figure 5.5 explique comment les tests sont ordonnancés avec les applications. L'ordonnancement de l'architecture est réalisé avec une période fixe qui est symbolisée sur la figure par les "ticks d'ordonnancement". Ainsi, à chaque "tick", trois fonctionnements sont possibles pour l'ordonnancement des tâches, selon qu'un processeur est testé, doit être testé ou non :

Si aucun processeur n'est et ne doit être testé : dans ce cas, tous les processeurs sont disponibles pour exécuter les applications. L'ordonnancement des tâches est donc réalisé sur (n) processeurs.

Si un processeur est en cours de test : dans ce cas, le processeur testé ne peut être utilisé pour l'application. L'ordonnancement des tâches est donc réalisé sur $(n-1)$ processeurs.

Si un processeur doit être testé : c'est à dire, tous les T_{test}/n. Dans ce cas, le choix du processeur à tester est réalisé par la politique d'ordonnancement des tests. Ce choix va déterminer l'efficacité du test et son impact sur les applications. Si le test des processeurs est réalisé sans prendre en compte l'état de ceux-ci, alors, le test peut avoir un impact important sur la durée d'exécution des applications. Si le processeur à tester exécute une application, alors elle devra être préemptée. Les politiques d'ordonnancement des tests que nous allons détailler dans la suite de cette partie, tentent de limiter l'impact sur les performances en prenant en compte l'état et la priorité des processeurs.

5.3.3 Les politiques d'ordonnancement des tests

L'objectif des paragraphes suivants est de présenter différentes implémentations de tests pseudo-périodiques et de tests strictement périodiques sur une architecture multiprocesseur. Nous voulons prouver que les tests pseudo-périodiques peuvent être utilisés pour détecter des erreurs intermittentes dans des architectures multiprocesseur, et qu'ils peuvent réduire l'impact des tests sur les applications par rapport à des tests périodiques à période fixe.

Pour cela, nous avons défini quatre politiques d'ordonnancement des tests. Deux ordonnancements implémentent des tests strictement périodiques (politiques *Aggressive* et *Aggressive with Pre-allocation*) et deux ordonnancements implémentent des tests pseudo-périodiques (politiques *Idleness Aware* et *Idleness and Priority Aware*). Les ordonnancements implémentant des tests strictement périodiques ont théoriquement l'impact le plus important sur les performances des applications, mais respectent le plus les périodes de test. Ces ordonnancements seront utilisés comme références pour évaluer le gain en performances obtenu par les deux approches pseudo-périodiques.

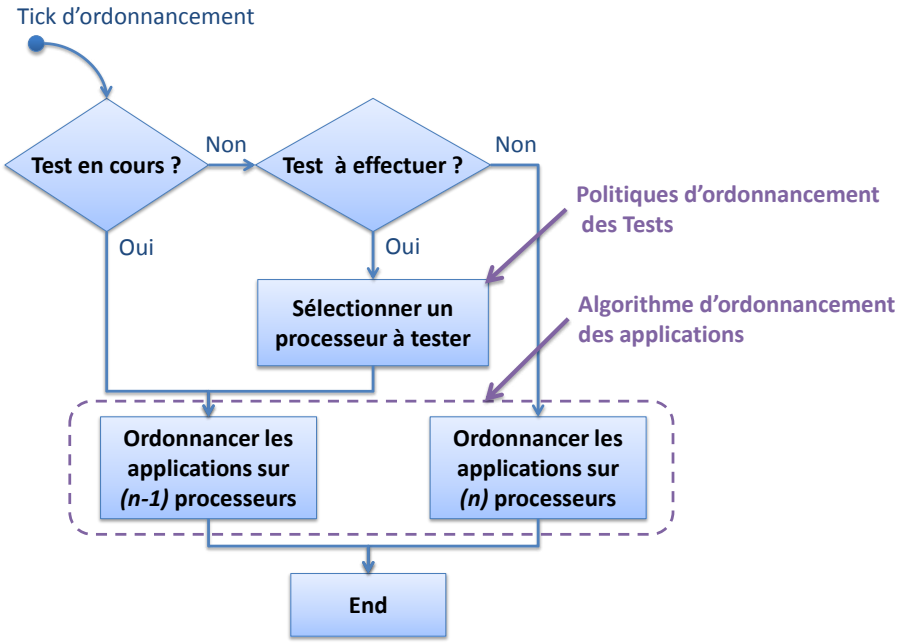


FIGURE 5.5 : *Ordonnancement des tests et des applications. À chaque "ticks d'ordonnancement", si aucun processeur n'est ou ne doit être testé alors l'ordonnancement des applications est réalisé normalement sur les (n) processeurs de l'architecture. Sinon il est réalisé sur (n-1) processeurs.*

Les approches de la littérature implémentent exclusivement des tests strictement périodiques, cependant, il existe des approches plus optimisées que d'autres. Dans ce sens, nous avons choisi d'implémenter deux approches de tests strictement périodiques, une première non optimisée et une deuxième bénéficiant d'optimisations au niveau de l'architecture. L'objectif est de confronter nos approches de tests pseudo-périodiques à des approches réalistes de tests strictement périodiques.

Politique Aggressive

La politique *Aggressive* (AP) implémente un test strictement prioritaire par rapport aux tâches de l'application. Les processeurs sont testés tour à tour dans un ordre prédéfini. Étant donné la liste L des processeurs à tester, l'algorithme parcourt la liste en partant du processeur 1 jusqu'au processeur n. Tous les temps T_{test}/n, le premier processeur non encore testé dans la liste L est sélectionné. Si une application est en cours d'exécution sur le processeur, celle-ci est préemptée pour y effectuer le test (voir figure 5.6).

L'algorithme 5.1 implémente cette politique. Pour chaque processeur dans l'architecture, si le processeur est "non testé", alors, il est sélectionné pour être testé. S'il est "testé", alors, un autre processeur est évalué. Si tous les processeurs sont "testés", alors, le test est réinitialisé et tous les processeurs sont mis dans un état "non testé".

La mise en place de cette politique a pour but d'étudier l'impact d'une allocation agressive sur le test et les applications, et de la comparer aux autres politiques. Cette politique d'allocation

a un coût important sur les performances, mais doit respecter au mieux la période de test. C'est l'approche généralement utilisée dans la littérature.

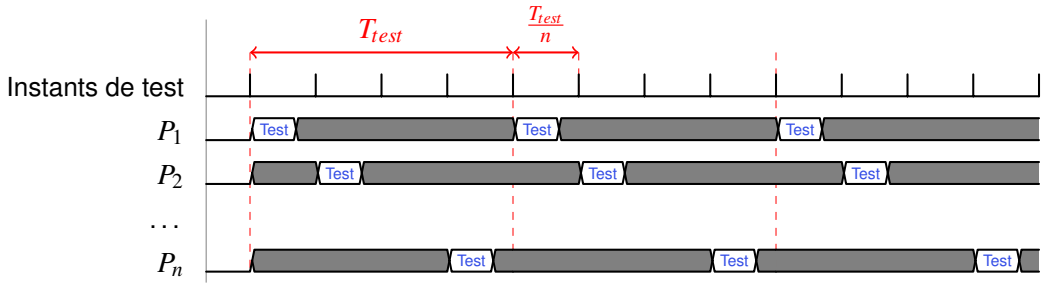


FIGURE 5.6 : *Exemple d'ordonnancement d'un test strictement périodique.*

```
For p=1 to Number_of_processor_in_architecture
Do
  If processor_not_tested(p) Then test(p) End_if
  Else_if all_processors_tested() Then init_tests()
  Else try_next_processor(p)
  End_if
End_for
```

Algorithme 5.1 – Implémentation de la politique Aggressive (AP)

Politique Idleness Aware

La politique *Idleness Aware* (IAP) exécute les tests sur les processeurs libres en priorité. Si aucun processeur à tester n'est libre alors ils sont testés dans un ordre prédéterminé. Cette politique prend en compte la disponibilité éventuelle des processeurs et limite ainsi le nombre de préemptions par rapport à la politique précédente. Cependant, elle peut induire des variations de la période de test sur chacun des processeurs. La figure 5.7 illustre ce phénomène et l'algorithme 5.2 implémente cette politique.

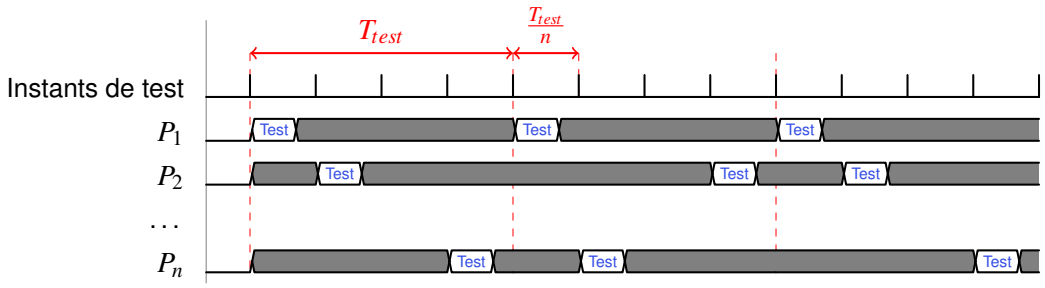


FIGURE 5.7 : *Exemple d'ordonnancement des tests non prioritaires. Ce type d'ordonnancement limite le nombre de préemptions des applications, mais induit des variations des périodes de test sur les processeurs (ici sur les processeurs P_2 et P_n).*

```
For p=1 to Number_of_processor_in_architecture
Do
  If processor_not_tested(p) Then
    If is_free(p) Then test(p)
    Else_if no_free_processor() Then test(p)
    End_if
  Else_if all_processors_tested() Then init_tests()
  Else try_next_processor(p)
  End_if
End_for
```

Algorithme 5.2 – Implémentation de la politique Idleness Aware (IAP)

Politique Idleness and Priority Aware

La politique *Idleness and Priority Aware* (IPAP) est identique à la politique précédente et exécute les tests sur les processeurs libres en priorité. La différence est, que si aucun processeur à tester n'est libre, alors le test est effectué sur le processeur ayant la plus faible priorité. Cette politique prend en compte la disponibilité éventuelle des processeurs et la priorité des tâches s'exécutant sur ceux-ci. Comme la politique précédente, elle peut induire des variations des périodes de test sur chacun des processeurs. L'algorithme 5.3 implémente cette politique.

```
For p=1 to Number_of_processor_in_architecture
Do
  If processor_not_tested(p) Then
    If is_free(p) Then test(p)
    Else_if no_free_processor() Then
      test_untested_processor_with_min_priority(p)
    End_if
  Else_if all_processors_tested() Then init_tests()
  End_if
End_for
```

Algorithme 5.3 – Implémentation de la politique Idleness and Priority Aware (IPAP)

Politique Aggressive with Pre-allocation

La politique *Aggressive with Pre-allocation* (APP) est identique à la politique *Aggressive* (AP) dans le choix des processeurs à tester. C'est à dire que les processeurs sont testés tour à tour dans un ordre prédéfini. Contrairement aux autres politiques, ici, le placement des tâches d'application est modifié. Le choix des processeurs à tester étant prédéfini, le but de cette politique est d'anticiper la préemption des applications en les plaçant en priorité sur les processeurs récemment testés. Dans la littérature, Li et al. [71] ont présenté un algorithme s'approchant de cette politique.

Cette politique tente de limiter au mieux le nombre de préemptions tout en conservant strictement les périodes de test. Pour cela, elle alloue les nouvelles tâches dans l'ordre inverse duquel les processeurs doivent être testés pour des erreurs. Ainsi, chaque nouvelle tâche est placée en priorité sur le dernier processeur libre qui a été testé. Remarquons que cette politique ne modifie en rien l'ordonnancement des tâches et qu'elle conserve les priorités des tâches. L'effet de cette politique est illustré sur la figure 5.8.

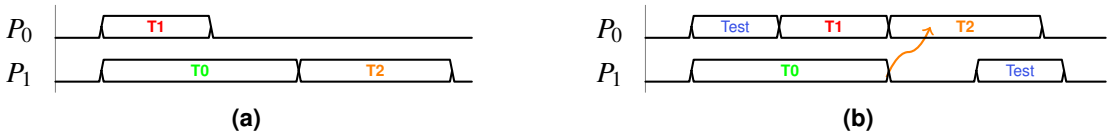


FIGURE 5.8 : *Illustration de la politique Aggressive with Pre-allocation. (a) Exemple d'allocations des tâches sans la présence de tests. (a) Après ajout des tests. Dans cet exemple, la tâche T2 est placée sur le processeur P0 en anticipation du test du processeur P1.*

5.4 Mise en place des simulations

Cette partie a pour but de présenter les moyens de simulation qui vont nous permettre de comparer les différentes politiques d'ordonnancement des tests. Nous montrerons comment modifier les paramètres de notre simulateur pour étudier l'impact du test périodique sur un large panel de cas d'études. En particulier, nous verrons comment modifier la charge des processeurs ainsi que le type d'application.

5.4.1 Objectifs et paramètres de simulation

L'objectif des simulations est de comparer les différentes politiques d'ordonnancement des tests. En particulier, nous voulons observer l'impact des politiques sur les applications et sur les tests. L'impact sur les applications sera observé par deux valeurs : le nombre de préemptions de tâches induit et l'augmentation de la durée des applications. L'impact sur le test sera observé par la probabilité de détection, défini dans le chapitre précédent.

Cependant, pour être pertinentes, les observations devront être faites pour différentes charges d'occupation de l'architecture et pour différentes applications. En effet, nous verrons que ces paramètres influencent fortement l'impact des politiques d'ordonnancement des tests. Nous verrons, dans la partie suivante, comment modifier l'application utilisée pour obtenir une charge d'occupation de l'architecture donnée.

5.4.2 Modification des paramètres de simulation

Dans le but d'analyser l'impact des tests sur les applications, nous souhaitons créer un large panel de conditions expérimentales. Cela concerne, la charge de l'architecture et le type d'appli-

cation utilisés (tâches longues ou courtes). Cette partie explique comment ces paramètres sont modifiés à partir de l'application présentée précédemment.

Taux d'occupation de l'architecture

Le lien entre les applications et le taux d'occupation peut être établi à l'aide de l'application de traitement d'image, présentée dans la partie précédente. Dans ce type d'application, la durée des tâches de traitement dépend directement des données (ici de l'image). Ainsi, la durée de chacune des tâches de traitement n'est pas nécessairement égale et cela a un impact sur le taux d'occupation de l'architecture. Pour une même application, en fonction de l'image à traiter, le taux d'occupation des processeurs varie.

Dans ce contexte, l'ajout de tâches de test en parallèle des applications n'aura pas le même impact en fonction des images à traiter, et donc en fonction des données d'entrée. En effet, plus le taux d'occupation de l'architecture est élevée et plus le coût du test est important. Ainsi, les simulations seront réalisées pour différents taux d'occupation de l'architecture. Cela sera réalisé en modifiant la durée des différentes tâches composant l'application définie précédemment.

Taux d'occupation d'un processeur : Soit t_{exec} le temps pendant lequel un processeur exécute une tâche, t_{sleep} le temps pendant lequel le processeur est au repos et t_{total} le temps total défini par $t_{total} = t_{exec} + t_{sleep}$. On peut alors définir le taux d'occupation To_i d'un processeur i par :

$$To_i = taux\ d'occupation\ processeur\ i = \frac{t_{exec}(i)}{t_{exec}(i) + t_{sleep}(i)} = \frac{t_{exec}(i)}{t_{total}(i)}$$

La figure 5.9 montre plusieurs taux d'occupation d'un processeur par rapport à une exécution standard.

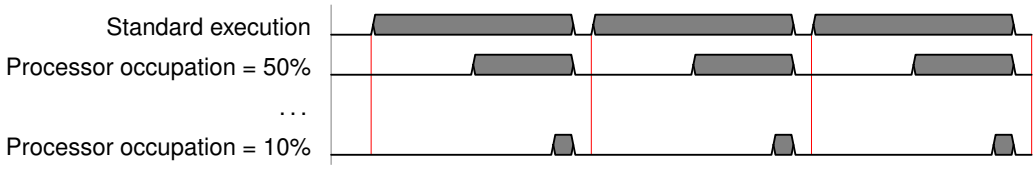


FIGURE 5.9 : *Modification de l'occupation des processeurs*

Taux d'occupation moyen de l'architecture : Le taux d'occupation moyen de l'architecture est défini en fonction du taux d'occupation des processeurs qui la composent. Soit To_n le taux d'occupation du processeur n et N le nombre total de processeurs dans l'architecture. On a alors :

$$\forall n \in [1;N]$$
$$To\_\text{sys} = taux\ d'occupation\ moyen\ architecture = \frac{\sum_{n=1}^{N} To_n}{N} \tag{5.1}$$

La figure 5.10 montre un exemple d'occupation d'une architecture composée de quatre processeurs. Dans cet exemple, les trois premiers processeurs ont une occupation de 100% et le dernier de 20%. Il en résulte une occupation de l'architecture de 80%.

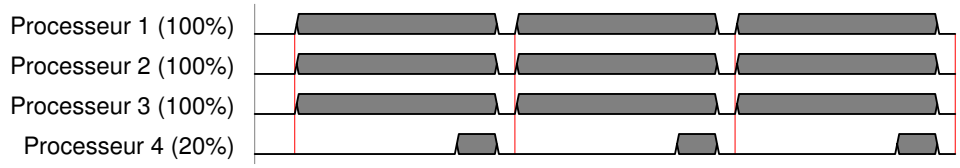


FIGURE 5.10 : *Exemple d'occupation de l'architecture. Dans cet exemple, le taux d'occupation est de* 80%.

Modification de l'application de traitement d'image : La modification du taux d'occupation de l'architecture peut être obtenue en modifiant la durée des tâches de traitement composant l'application de labelling (tâches T1 à T8 sur la figure 5.3). Dans la réalité, la durée des tâches évolue en fonction des données d'entrée, c'est à dire, en fonction de l'image à traiter. Dans notre cas, les tâches T0, et les tâches supérieures à T9, ont une durée d'exécution très inférieure aux autres, ces tâches n'auront donc que peu d'influence sur le taux d'occupation. La figure 5.11 illustre un exemple d'ordonnancement typique de l'application. Dans cet exemple, la durée de la tâche T8 est deux fois plus faible que les autres tâches de traitement. Dans ce cas, le taux d'occupation est de $To\_\text{sys} = (7*100+50)/8 = 93,75\%$. Ainsi, la modification du taux d'occupation de l'architecture sera réalisée sur le même principe. Un taux d'occupation de 100% signifiera que la durée des huit tâches de traitement est identique. Mais en réalité, le taux d'occupation de l'architecture peut rarement atteindre cette valeur.

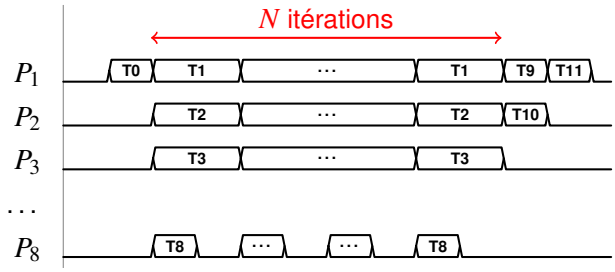


FIGURE 5.11 : *Modification de l'application de traitement d'images. Dans cet exemple, la durée de la tâche T8 est deux fois plus faible que les autres tâches de traitement. Dans ce cas, le taux d'occupation est* $To\_sys = (7*100+50)/8 = 93,75\%$.

Modification du type d'application

Pour une durée d'application et un taux d'occupation de l'architecture donnés, l'impact du test sur la durée de l'application peut être différent. En effet, le nombre de préemptions induites par le test diffère suivant la durée des tâches (courte ou longue). Plus une tâche est longue et plus sa probabilité d'être préemptée plus d'une fois est importante. Or, toute préemption implique un coût pour l'application. Lors de la préemption d'une tâche, ses données et l'état du processeur

doivent être sauvegardés. Ainsi, plus le nombre de préemptions est important et plus la durée totale de l'application est allongée.

Dans notre cas, nous avons choisi d'étudier deux types d'applications. La première application est composée de tâches longues : huit tâches de 125 ms (figure 5.12a). La deuxième application est composée de tâches courtes : cent tâches de 10 ms (figure 5.12b). Le but de ces deux applications est de comparer l'impact d'un même test sur le nombre de préemption et la durée de l'application.

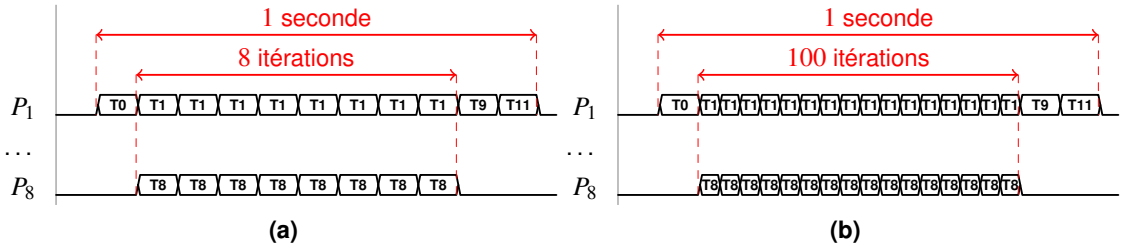


FIGURE 5.12 : *Modification du type d'application. La durée d'exécution de l'application est constante d'une seconde. Deux types d'applications sont identifiés : (a) tâches longues et (b) tâches courtes.*

5.5 Résultats

L'objectif de ce chapitre est de comparer les différentes implémentations de tests pseudo-périodiques, aux tests strictement périodiques. Pour rappel, les différentes politiques ordonnancement des tests sont :

AP : la politique *Aggressive* est un test strictement périodique. Les processeurs sont testés tour à tour dans un ordre prédéfini.

IAP : la politique *Idleness Aware* implémente un test pseudo-périodique, elle exécute les tests sur les processeurs libres en priorité. Si aucun processeur à tester n'est libre alors ils sont testés dans un ordre prédéterminé.

IPAP : la politique *Idleness and Priority Aware* implémente un test pseudo-périodique, elle exécute les tests sur les processeurs libres en priorité. Si aucun processeur à tester n'est libre alors le test est effectué sur le processeur non testé ayant la plus faible priorité.

APP : la politique *Aggressive with Pre-allocation* est identique à la politique AP, mais évite de placer les tâches d'application sur les processeurs à tester.

Les simulations présentées dans cette partie sont réalisées pour des taux d'occupation de l'architecture allant de 30% à 100% (voir § 5.4). Pour chaque valeur de charge, deux durées de tâches de test sont considérés : 125 ms et 10 ms.

Les différentes politiques d'ordonnancement des tests sont comparées par rapport à leur impact sur la durée d'exécution des applications et sur la probabilité de détection du test. Avant de présenter ces résultats, nous détaillons la configuration du test périodique utilisé pour les simulations.

5.5.1 Configuration du test périodique

Nous avons montré dans le chapitre 4 que la probabilité de détection d'un test périodique dépend de sa période, de l'erreur à détecter, et de la fenêtre d'observation du test. Ici, l'évaluation de la probabilité de détection sera faite à la fin de l'application. Sans aucun test, la durée d'exécution de l'application est d'une seconde.

Nous avons choisi de détecter une erreur intermittente ayant les caractéristiques suivantes : $\mu/\lambda = 10$ et $\mu = 0.1$ ms^{-1}. Dans ce cas, la période de test pour obtenir une probabilité de détection de $99,9\%$ après une seconde est $T_{test} = 7,69$ ms. Cela représente 130 tests sur chaque processeur, ainsi, un nouveau processeur sera testé toutes les $T_{test}/n = 961,53$ us.

Le dernier élément à prendre en compte dans la définition du test périodique est la durée des tests. Nous avons sélectionné une méthode de test de type Software-Based Self-Test (SBST) pour le test des processeurs. Or, Gizopoulos et al. [66] ont montré que le temps de test d'un MIPS32 par ces méthodes est d'environ $0,2$ ms. Cependant, nous souhaitons étudier l'impact de la longueur du test sur son coût. Dans ce sens et afin de conserver cet ordre de grandeur, nous avons choisi d'utiliser deux longueurs de test : $l1 = 0,15$ ms et $l2 = 0,3$ ms.

5.5.2 Impact des politiques d'ordonnancement sur les applications

Cette partie compare les politiques d'ordonnancement des tests par rapport à leur impact sur le nombre de préemptions et l'augmentation du temps d'exécution induits par les tests.

Nous considérons ici que l'application est originellement conçue pour être exécutée seule sur l'architecture. C'est à dire que sans les tâches de test, l'application ne subit aucune préemption. Néanmoins, l'ajout d'un test périodique cause nécessairement des préemptions. De manière explicite, il y a préemption si un test doit être exécuté sur un processeur qui exécute déjà une tâche. Et de manière implicite, les tâches de l'application peuvent se préempter entre elles, en fonction des contraintes temps-réel. Ainsi, le nombre de préemption indiqué par les figures 5.13a et 5.13b représente le nombre total de préemptions observées à la fin de l'exécution de l'application et sur l'ensemble des processeurs.

La figure 5.13a présente le nombre de préemptions observées sur une application constituée de tâches longues (125 ms), alors que la figure 5.13b présente ce nombre pour une application constituée de tâches courtes (10 ms). De même, la figure 5.14a présente l'augmentation de la durée de l'application observée sur une application constituée de tâches longues (125 ms), alors que la figure 5.14b présente ce nombre pour une application constituée de tâches courtes (10 ms).

Observation générale : De manière générale, nous constatons que plus le taux d'occupation de l'architecture est important et plus l'ajout du test induit des préemptions. Ce phénomène est indépendant de la durée du test et du type d'application. Néanmoins, la pente des courbes semble plus prononcée autour de 90% de taux d'occupation de l'architecture. Cela est plus visible sur l'application constituée de tâches longues.

Plus exactement, un taux d'occupation inférieur à $87,5\%$ équivaut à une durée d'exécution nulle pour une des huit tâches de traitement. Et plus généralement, pour n processeurs, cette limite apparait pour un taux d'occupation de $(n-1)/n$. Au delà de cette limite, l'impact du test sur le nombre de préemptions augmente avec la taille des tâches. Pour l'application constituée de tâches longues, cette augmentation est d'environ 800 préemptions pour 10% d'occupation et elle est divisée par deux pour l'application constituée de tâches courtes.

Concernant l'augmentation de la durée des applications, les mêmes observations peuvent être effectuées : l'augmentation de la durée d'exécution est accélérée pour un taux d'occupation supérieur à $87,5\%$. De plus, dans le cas d'applications constituées de tâches longues, le coût du test sur la durée d'exécution reste inférieure à 2% pour un taux d'occupation inférieur à 90%, mais augmente à plus de 10% dès 95% de taux d'occupation.

Concernant la durée des tests, nous ne constatons aucune différence sur l'application constituée de tâches longues. Cependant, sur l'application constituée de tâches courtes, la différence de coût induit par le test est d'environ 2%.

Comparaison des politiques : Quelle que soit la durée des tests ou le type d'application, la politique AP impose toujours le plus grand nombre de préemptions. En comparaison, les poli-

tiques IAP et IPAP affichent entre 30% et 50% de préemptions en moins, pour une application composée de tâches courtes, et entre 10% et 30% pour une application composée de tâches longues. Ces résultats confirment qu'il est préférable d'utiliser des politiques non prioritaires pour limiter le nombre de préemptions.

Concernant les politiques IAP et IPAP, nous constatons que sur une application composée de tâches longues, il n'y a pas de différence, quel que soit le taux d'occupation de l'architecture. Cependant, pour une application composée de tâches courtes, la politique IPAP implique 10% de préemptions en moins pour des taux d'occupation importants.

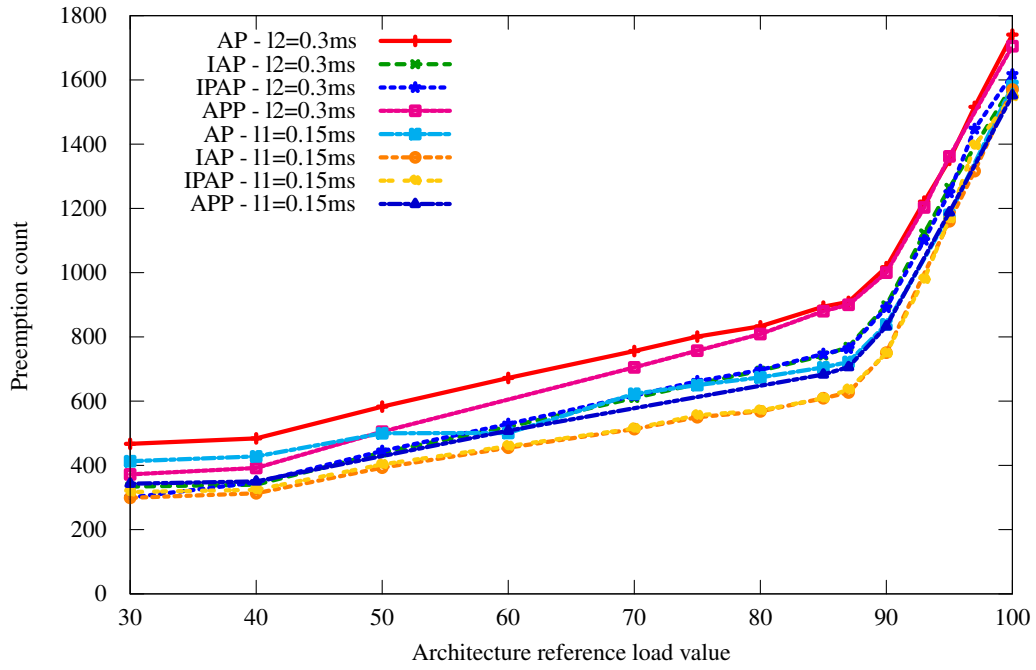
La politique APP est une politique agressive de même que la politique AP. Cependant, la méthode de pré-allocation des tâches mise en œuvre dans cette politique semble être efficace pour réduire le nombre de préemptions. En effet, nous constatons une réduction du nombre de préemptions de 50% par rapport à la politique AP, mais seulement pour des taux d'occupation faibles. A partir d'un taux d'occupation de 85%, cette politique est identique à la politique AP.

Concernant l'augmentation de la durée des applications, dans le cas d'applications constituées de tâches longues, il n'y a pas de différence entre les différentes politiques tant que le taux d'occupation est inférieur à 87.5%. Cependant, après cette limite, la différence entre les politiques agressives (AP et APP) et les politiques non prioritaires (IAP et IPAP) est de 1% à 3%.

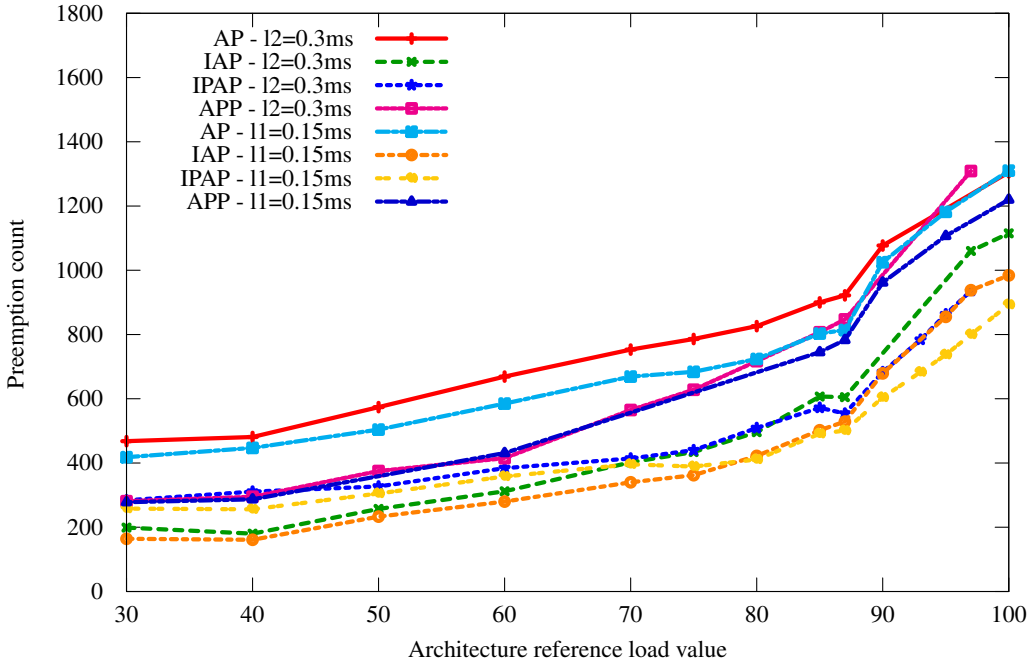
Dans le cas d'applications constituées de tâches courtes, la différence entre les politiques est plus notable. La politique AP reste la plus coûteuse pour l'application. Concernant les autres politiques, nous constatons peu de différences pour un taux d'occupation inférieur à 87.5%. Cependant, après cette limite, la politique agressive APP impose le même coût que la politique AP. La différence avec les autres politiques (IAP et IPAP) est alors comprise entre 2% et 4% suivant la longueur des tests.

Conclusion : Les résultats présentés confirment que l'utilisation de politiques pseudo-périodiques (IAP et IPAP) permet de diminuer le nombre de préemptions jusqu'à 50% par rapport aux politiques strictement périodiques (AP et APP).

Comme pour le nombre de préemptions, l'augmentation de la durée d'exécution induite par les tests reste toujours plus faible avec l'utilisation de politiques pseudo-périodiques (de 1% à 6%).

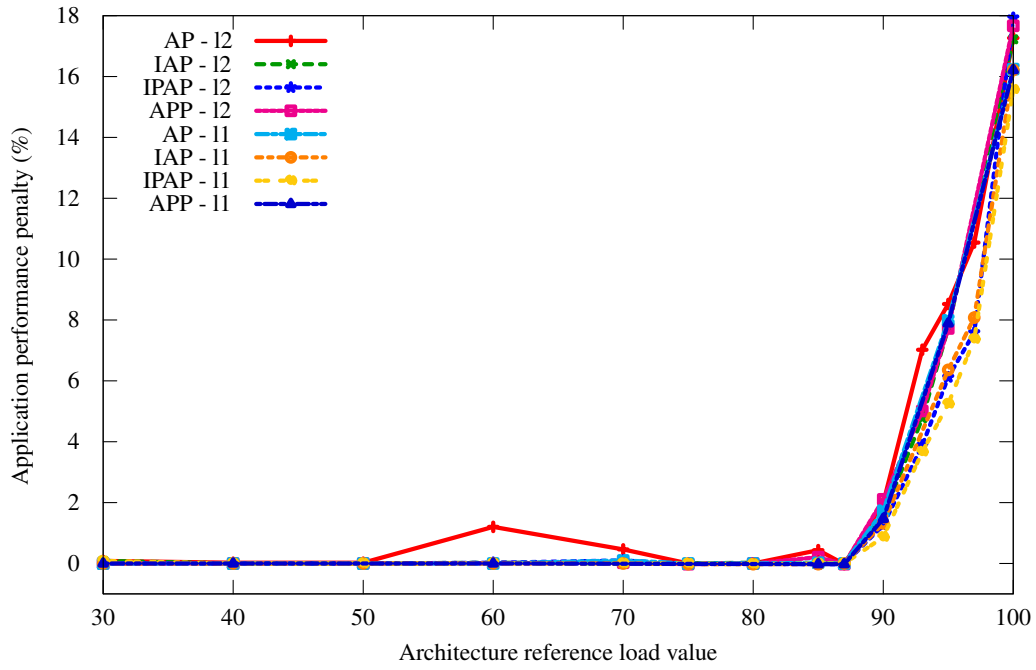


(a) *Application constituée de tâches longues (*125 *ms).*

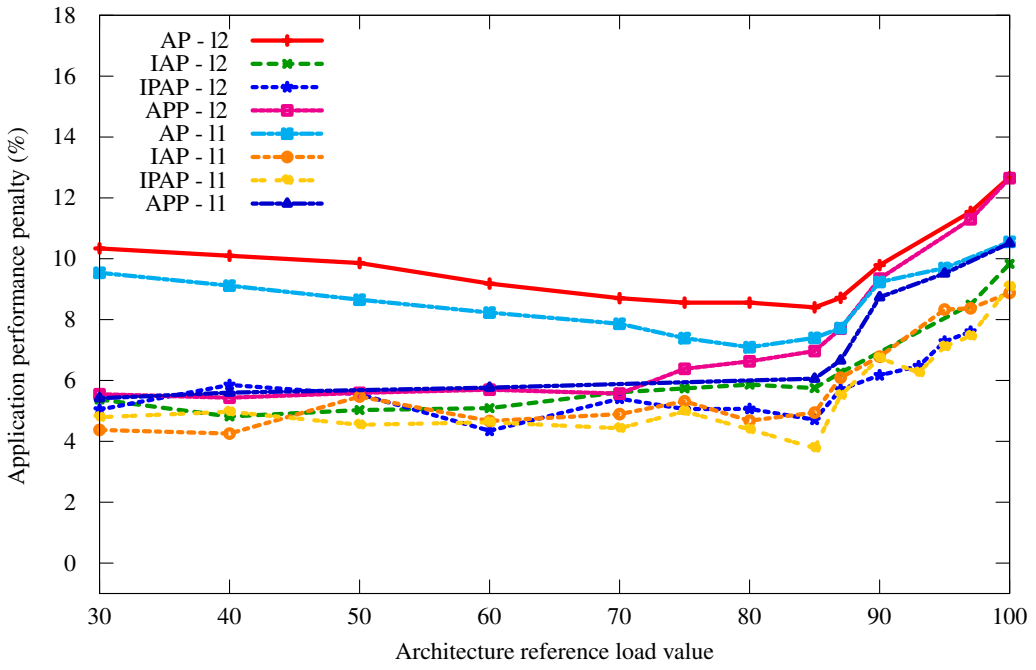


(b) *Application constituée de tâches courtes (*10 *ms).*

FIGURE 5.13 : *Nombre de préemptions induit par les politiques d'ordonnancement des tests : (a) pour les tâches longues (*125 *ms) et (b) pour les tâches courtes (*10 *ms).*



(a) *Application constituée de tâches longues (*125 *ms).*



(b) *Application constituée de tâches courtes (*10 *ms).*

FIGURE 5.14 : *Augmentation de la durée de l'application induite par les politiques d'ordonnancement des tests : (a) pour les tâches longues et (b) pour les tâches courtes.*

5.5.3 Impact des politiques d'ordonnancement sur la probabilité de détection du test

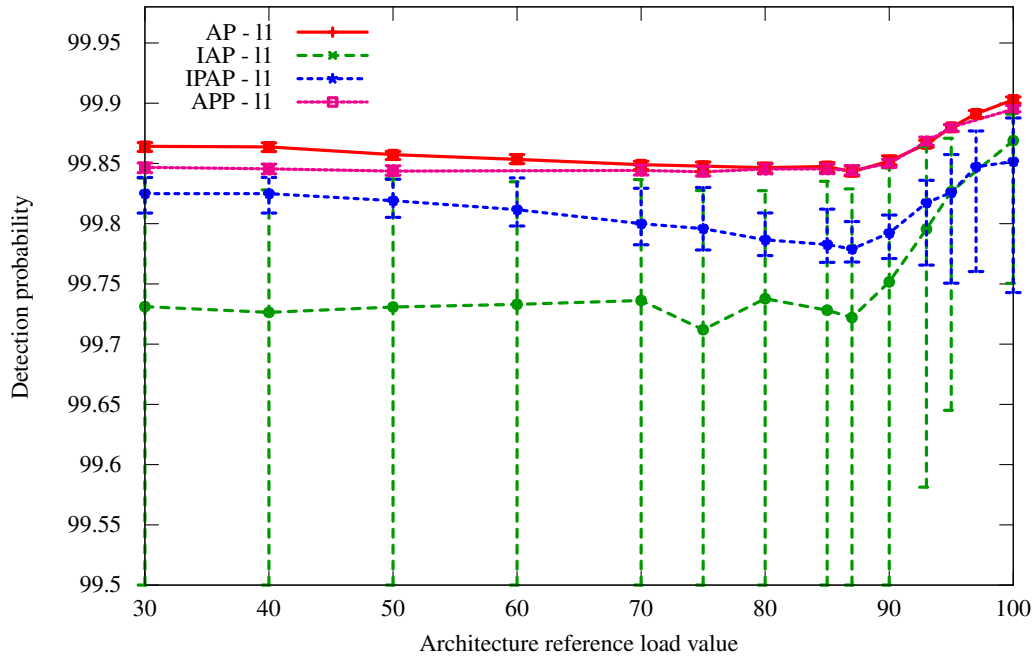
L'objectif des tests pseudo-périodiques est de détecter des erreurs intermittentes. Afin de les implémenter dans une architecture multiprocesseur, nous avons défini plusieurs politiques d'ordonnancement des tests. De par leur conception, nous savons que certaines politiques conservent les périodes de tests (AP et APP), à la différence des autres (IAP et IPAP). Or, nous avons montré dans le chapitre précédent que si les intervalles de tests ne sont pas constants, alors cela peut nuire à la détection des erreurs intermittentes. Afin de mesurer et de comparer l'efficacité du test pour chacune des politiques, nous avons défini une métrique : la probabilité de détection.

Cette probabilité est calculée à la fin de l'exécution de l'application, sur chacun des processeurs. Ainsi, nous avons pour chaque valeur de taux d'occupation autant de probabilités de détection que de processeurs dans l'architecture. La moyenne de ces valeurs nous donne alors la probabilité de détection moyenne sur toute l'architecture. Dans ce sens, les courbes 5.15a et 5.15b présentent, d'une part, la probabilité de détection moyenne et, d'autre part, les probabilités maximales et minimales observées sur tous les processeurs. Cette dernière valeur nous permettra de comparer la dispersion de la probabilité de détection induite par les politiques d'ordonnancement de tests. L'objectif étant de montrer que les politiques pseudo-périodiques peuvent conserver une probabilité de détection suffisante sur tous les processeurs.

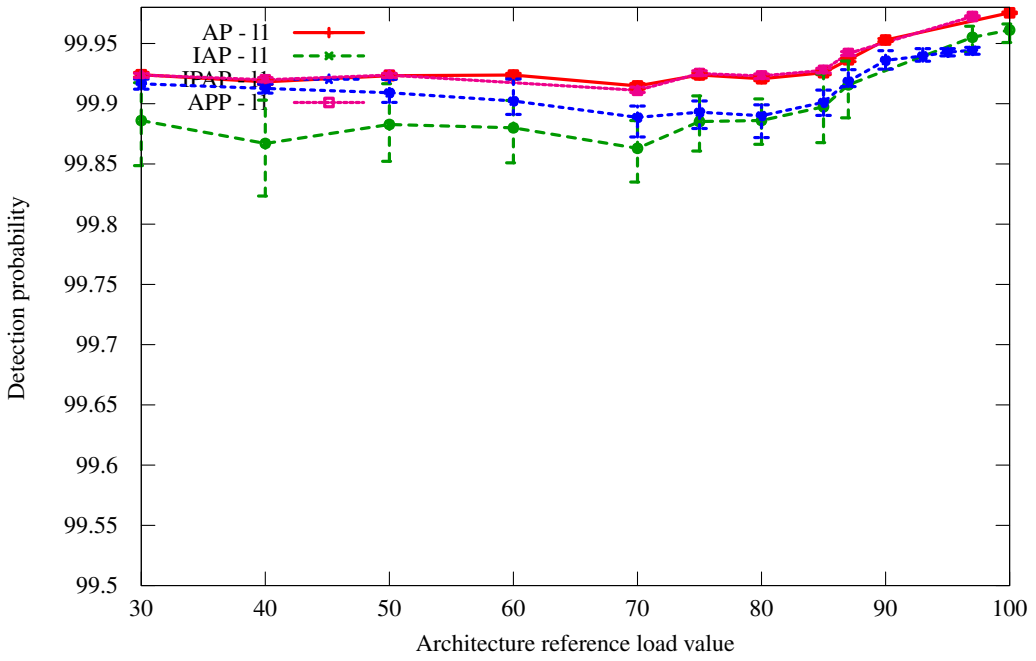
Observation générale : Nous avons vu que la probabilité de détection est calculée à la fin de l'exécution de l'application. Ainsi, elle évolue avec la durée de l'application. Plus l'exécution de l'application est longue et plus la probabilité de détection est importante. Ceci est à mettre en avant dans l'interprétation des courbes 5.15a et 5.15b. En effet, l'évolution de probabilité de détection est à corréler avec l'augmentation de la durée d'exécution de l'application. Dans ce sens nous nous intéressons ici uniquement à la variation de la probabilité de détection sur les processeurs de l'architecture.

Comparaison des politiques : Sans surprise, nous constatons que les politiques agressives AP et APP offrent le moins de dispersion de la probabilité de détection sur les processeurs. A l'inverse, les politiques IAP et IPAP induisent une dispersion due à une distribution non homogène des intervalles de test sur les processeurs. Cela signifie que l'efficacité du test n'est pas la même sur tous les processeurs. D'une manière générale, cette dispersion est plus importante dans le cas d'applications constituées de tâches longues. Dans ce type d'application l'écart peut être supérieur à $0,3\%$ dans le cas de la politique IAP, et inférieur à $0,1\%$ dans le cas de la politique IPAP. Alors que pour des applications constituées de tâches courtes, la politique IAP induit une dispersion de seulement $0,08\%$ et la politique IPAP $0,05\%$. Nous avons ici représenté les résultats pour une durée de test $l1 = 0,15$ ms, néanmoins, les conclusions sont identiques pour $l2 = 0,3$ ms.

Conclusion : Les politiques agressives AP et APP offrent le moins de dispersion de la probabilité de détection. Cependant, dans tous les cas étudiés, la politique IPAP induit un écart maximal de $0,1\%$ entre les probabilités de détection maximales et minimales observées sur tous les processeurs. Ainsi, cette politique pseudo-périodique conserve de bonnes propriétés de détection des erreurs dans toutes les conditions étudiées et sur tous les processeurs.



(a) *Application constituée de tâches longues (*125 *ms).*



(b) *Application constituée de tâches courtes (*10 *ms).*

FIGURE 5.15 : *Probabilité de détection induite par les politiques d'ordonnancement des tests : (a) pour les tâches longues et (b) pour les tâches courtes. Ces courbes présentent, d'une part, la probabilité de détection moyenne et, d'autre part, les probabilités maximales et minimales observées sur tous les processeurs.*

5.5.4 Synthèse et mise en valeur des tests pseudo-périodiques

Les résultats présentés dans cette section comparent plusieurs implémentations de tests pseudo-périodiques avec des tests strictement périodiques. D'une part, ces résultats comparent l'impact des politiques d'ordonnancement des tests sur les performances des applications, en observant le nombre de préemptions et l'augmentation de la durée d'exécution de l'application. D'autre part, ils comparent l'efficacité des tests pour la détection des erreurs intermittentes, en observant la probabilité de détection des tests et sa dispersion sur les processeurs. La politique idéale est celle qui induit le moins de pénalités de performances sur l'application et qui conserve une faible dispersion de la probabilité de détection sur les processeurs. À la vue des résultats, la politique pseudo-périodique IPAP, prenant en compte les processeurs au repos et la priorité des tâches, offre le meilleur compromis entre performance et probabilité de détection.

En observant le nombre de préemptions et l'augmentation de la durée d'exécution de l'application, nous avons constaté que l'utilisation de tests pseudo-périodiques (IAP et IPAP) permet de réduire de 50% le nombre de préemptions et de 1% à 6% la durée d'exécution, par rapport aux tests strictement périodiques (AP et APP) et dans toutes les conditions d'exécution étudiées. De plus, même si la politique strictement périodique APP induit un impact sur les performances plus faibles que la politique AP – en particulier si le taux d'occupation de l'architecture est inférieur à $87,5\%$ pour huit processeurs – elle reste toujours plus coûteuse que les approches pseudo-périodiques.

Concernant la probabilité de détection, de par leur conception ce sont les tests strictement périodiques (AP et APP) qui induisent la dispersion la plus faible sur les processeurs. Cependant, la politique pseudo-périodique IPAP induit un écart maximal de seulement $0,1\%$ entre les probabilités de détection maximales et minimales observées sur tous les processeurs. Ainsi, cette politique conserve de bonnes propriétés de détection des erreurs, sur tous les processeurs et dans toutes les conditions étudiées. Nous confirmons ainsi l'intérêt des tests pseudo-périodiques.

Pour aller plus loin

Spécificités de l'architecture : Les résultats présentés dans ce chapitre ont été obtenus avec l'architecture SCMP. Ainsi, ils ont bénéficiés de plusieurs caractéristiques spécifiques de cette architecture. En particulier, cette architecture bénéficie d'un ordonnancement dynamique et de mécanismes optimisés pour la préemption et la migration des tâches. Ainsi, l'impact du test sur la durée d'exécution des applications est minimisé.

Si l'ordonnancement des tâches était statique, alors, chaque tâche préemptée par un test devrait attendre la fin du test pour continuer son exécution. Au contraire, dans notre cas, quand une tâche est préemptée, elle peut continuer son exécution sur un autre processeur en parallèle du test. Ainsi, dans le cas d'un ordonnancement statique, le coût du test sur la durée des applications serait plus important, surtout pour des tests strictement périodiques. En effet, le coût des préemptions serait plus important, ce qui creuserait d'autant plus l'écart entre les tests strictement périodiques et les tests pseudo-périodiques.

Vers une auto-adaptation du test en ligne

Nous avons montré dans ce chapitre que les tests pseudo-périodiques offrent le meilleur compromis entre performance et probabilité de détection, en prenant en compte les processeurs au repos et la priorité des tâches. Pour compléter cette étude, l'ordonnancement des tests et des tâches applicatives pourrait être adapté en fonction d'informations de l'architecture. La figure 5.16 montre cette adaptation.

En particulier, nous avons vu dans le chapitre 2 que l'activité des processeurs a un impact sur leur vieillissement. Or, les politiques d'ordonnancement des tests définies dans ce chapitre n'en tiennent pas compte. Ainsi, les politiques pourraient être modifiées, pour adapter la probabilité de détection des tests à l'activité des processeurs. Par exemple, nous avons vu que les politiques pseudo-périodiques induisent – même si elle est faible – une dispersion de la probabilité de détection sur les processeurs. Ainsi, le processeur ayant l'historique d'activité le plus grand devrait être celui pour lequel la période de test est la mieux respectée. Ainsi, les algorithmes d'ordonnancement des tests peuvent être complétés en ajoutant des informations de l'architecture, comme l'historique de l'activité des processeurs, de la température, de la consommation ou des taux d'erreurs.

De plus, le chapitre 2 met en évidence la régénération des processeurs après une période d'inactivité. Cela peut être pris en compte par le contrôle de l'architecture, qui peut décider de stopper un processeur soumis à des erreurs intermittentes. Ainsi, l'architecture bénéficierait d'un système complet de détection et de gestion des erreurs intermittentes.

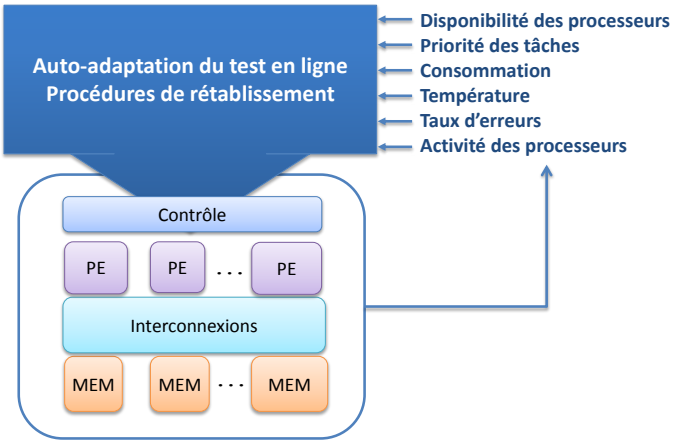


FIGURE 5.16 : *Auto-adaptation du test en ligne en utilisant des informations de l'architecture. De même, le contrôle peut réaliser le rétablissement des processeurs en les stoppant.*

5.6 Conclusion

Les objectifs de ce chapitre étaient de présenter l'implémentation d'un test pseudo-périodique sur une architecture multiprocesseur ; et de confirmer dans des conditions réelles d'exécution, que les tests pseudo-périodiques permettent de réduire l'impact sur les performances, comparés à des tests strictement périodiques. Il n'existe aujourd'hui, à notre connaissance, aucune implémentation de tests pseudo-périodiques sur une architecture multiprocesseur, capable de détecter des erreurs intermittentes. Ainsi, cette étude tente d'y apporter une solution.

Pour répondre à ces objectifs, nous avons présenté dans la première partie différentes implémentations d'ordonnancement des tests intégrées dans le contrôle de l'architecture. Ainsi, deux politiques strictement périodiques et deux politiques pseudo-périodiques ont été définies. Généralement, seules les politiques strictement périodiques sont utilisées or, elles ne tiennent pas compte des tâches exécutées sur les processeurs ; alors que les politiques pseudo-périodiques prennent en compte les processeurs au repos et la priorité des tâches.

Afin de comparer ces politiques, nous avons réalisé plusieurs simulations nous permettant d'observer les différentes politiques dans différentes conditions, de taux d'occupation de l'architecture, et de types d'applications. Cela nous a permis de comparer ces politiques en fonction, du nombre de préemptions, de l'augmentation de la durée d'exécution de l'application et de la probabilité de détection induits par les tests. Nous avons ainsi confirmé qu'une politique pseudo-périodique prenant en compte les processeurs au repos et la priorité des tâches offre le meilleur compromis entre performance et probabilité de détection.

Conclusions et perspectives

Alors que les fautes intermittentes suscitent un intérêt grandissant dans le domaine des circuits intégrés, encore peu d'études existent actuellement sur ce sujet. De plus, aucune technique de la littérature n'est adaptée à leur détection dans les architectures multiprocesseur. Afin d'apporter une solution à ces problèmes, ce mémoire propose d'une part, une plateforme expérimentale permettant de mettre en évidence les erreurs intermittentes dans les circuits intégrés ; et d'autre part, il propose une méthode de test pseudo-périodique adaptée à la détection des erreurs intermittentes dans les architectures multiprocesseur.

Synthèse des contributions

Le premier chapitre de ce mémoire nous a permis de définir, dans un premier temps, les principaux concepts de fiabilité qui sont utilisés au cours de ce mémoire, et dans un deuxième temps, les problèmes de fiabilité affectant les systèmes sur puce. En particulier, certains défauts, non visibles au début de la vie du circuit intégré, peuvent se matérialiser par des fautes de délais avec la combinaison du vieillissement du circuit, de fluctuations des tensions d'alimentation et de la température. Nous avons ainsi montré que, avant que le système ne soit défaillant - pendant sa période de vie utile - des fautes intermittentes peuvent se manifester.

Cependant, nous ne disposons que de peu d'études sur ce type de fautes. En particulier, jusqu'à présent, aucune étude n'a observé des fautes intermittentes sur un système embarqué dans une technologie actuelle. Afin de palier ce manque d'informations, le deuxième chapitre de ce mémoire définit une plateforme expérimentale capable d'observer des erreurs intermittentes. Pour cela, des processeurs en technologie 65 nm sont soumis, d'une part à une température supérieure à 160°C et d'autre part à l'exécution d'un ensemble d'applications. Nous avons pu confirmer que les erreurs intermittentes peuvent être observées avant l'apparition de fautes permanentes. De plus, elles peuvent apparaître très tôt avant la période d'usure du circuit. Concernant le mode d'apparition des erreurs intermittentes, toutes nos observations ont montré une apparition de type *burst*. Les erreurs apparaissent en rafale et seul l'arrêt des processeurs semble les stopper. Par cette étude, nous confirmons qu'il est nécessaire de mettre en place des méthodes de détection des erreurs intermittentes dans les circuits intégrés très submicroniques.

Nous savons que les évolutions technologiques permettent d'augmenter la densité d'intégration des circuits intégrés. De plus, seules les architectures multiprocesseurs semblent pouvoir suivre l'évolution des applications en apportant le bon compromis entre performance et consom-

mation. Pour ces raisons, nous avons choisi de rechercher des techniques de détection des erreurs intermittentes adaptées à ce type d'architecture. Dans ce sens, le troisième chapitre de ce mémoire réalise un état de l'art des méthodes de détection en ligne des erreurs. L'objectif étant de sélectionner la méthode la mieux adaptée à nos critères. Nous avons conclu que seules des méthodes de détection basées sur l'utilisation de structures de test pourraient convenir, si elles sont utilisées au sein d'un test périodique.

L'utilisation d'un test périodique a généralement un impact important sur la durée d'exécution des applications. Ainsi, le quatrième chapitre étudie la faisabilité d'utiliser un test pseudo-périodique pour détecter les erreurs intermittentes. En effet, contrairement aux approches utilisées dans la littérature, nous pensons que le test en ligne ne doit pas nécessairement être prioritaire et donc strictement périodique. Pour cela, nous avons montré qu'il est possible de modéliser les erreurs intermittentes à l'aide d'un modèle de Markov. Cela nous a permis de décrire la probabilité de détection du test, en fonction des caractéristiques des erreurs à détecter et des intervalles de test. Finalement, nous avons conclu que l'utilisation d'un test pseudo-périodique est possible. Par exemple, si une erreur de $0,04\%$ est tolérée sur la probabilité de détection alors le système peut tolérer des variations des intervalles de test jusqu'à 40% de la période de test initiale.

Afin de compléter cette étude théorique, le cinquième chapitre de ce mémoire présente et compare plusieurs implémentations d'un test périodique sur une architecture multiprocesseur. Pour cela, nous avons modifié le contrôle de l'architecture, afin de simuler plusieurs politiques d'ordonnancement des tests. Nous avons ainsi comparé des politiques strictement périodiques avec des politiques pseudo-périodiques, dans différentes conditions de taux d'occupation de l'architecture et de types d'applications. Cela nous a permis de comparer ces politiques en fonction du nombre de préemptions, de l'augmentation de la durée d'exécution de l'application et de la probabilité de détection induits par les tests. Les résultats des différentes simulations confirment qu'une politique pseudo-périodique prenant en compte les processeurs au repos et la priorité des tâches offre le meilleur compromis entre performance et probabilité de détection.

Perspectives

Les perspectives principales de ce mémoire viennent sans doute de la partie expérimentale. En effet, les expériences sont souvent compliquées à mettre en œuvre et les résultats sont rarement ceux que l'on s'attendait à trouver. Et l'étude des fautes intermittentes ne déroge pas à la règle. En effet, notre étude étant la première à s'intéresser à ce type de fautes de manière expérimentale, aucune étude actuelle n'a pu nous aider à mettre en place notre protocole expérimental. Ainsi, ce mémoire tente de combler ce manque. Cependant, cette première étude laisse la place à de nombreuses évolutions. Pour cela, plusieurs voies peuvent être envisagées afin de préciser les conditions expérimentales, confirmer ou infirmer nos observations et conjectures, et effectuer de nouvelles observations.

Perspectives à court terme

Préciser les conditions expérimentales Notre étude expérimentale se base sur une combinaison de phénomènes comme source des fautes intermittentes. En effet, d'après la littérature, nous avons pu déterminer que ces fautes apparaissent par la combinaison du vieillissement du circuit, avec des fluctuations des tensions d'alimentation et de la température. Cependant le seul vecteur de stress utilisé dans nos expériences est la température, ainsi l'impact de variations de la tension d'alimentation doit, aussi, être évalué.

De plus, nos expériences souffrent d'incertitudes au niveau de la mesure des conditions expérimentales. En effet, pour être plus précise, la température doit être mesurée à l'intérieur du circuit intégré. De même, pour les variations de la tension d'alimentation des processeurs, celles-ci doivent être mesurées au plus près des processeurs.

A court terme, les composants que nous avons utilisé dans nos expériences (FPGA Xilinx Virtex5FX) peuvent convenir s'ils sont intégrés sur une carte spécifique. Ainsi, la conception d'une carte électronique est nécessaire.

Effectuer de nouvelles observations Afin de confirmer les tendances que nous avons mises en évidence, de nouvelles observations doivent être réalisées sur des composants dans des technologies différentes et plus récentes. En effet, toutes nos observations sont dépendantes de la technologie utilisée. Or, cette plateforme expérimentale a mis en évidence le mode d'apparition particulier des fautes intermittentes. En effet, ces fautes apparaissent toujours en *bursts*, c'est à dire avec une période et une durée données, ce qui les différencient des autres types de fautes. Cette observation est cohérente avec la littérature scientifique mais doit être confirmée.

En observant les conditions de disparition des fautes intermittentes, nous avons constaté, dans notre cas, que l'arrêt du processeur suffit à stopper les erreurs. Ce résultat peut permettre d'envisager des procédures de rétablissement des processeurs. Cependant, des informations supplémentaires sont nécessaires. Pour cela, de nouvelles expériences doivent être mises en œuvre pour décrire au mieux ce phénomène. En particulier, l'impact du temps de repos des processeurs doit être analysé plus en détails. Cela peut être réalisé avec la plateforme actuelle sans modification.

Perspectives à plus long terme

Vers une puce d'étude Les perspectives apportées par la réalisation d'une puce d'étude sont multiples. Premièrement, nos objectifs concernent les architectures multiprocesseur, or, nos expériences ont été réalisées à partir de composants monoprocesseur embarqués au sein de composants FPGA. Ainsi, il est nécessaire de valider toutes nos observations sur une architecture multiprocesseur de type ASIC. Deuxièmement, tous les moniteurs nécessaires à notre étude pourraient être intégrés au plus près des processeurs. Cela permettrait d'obtenir des observations plus précises et plus fiables. Outre les moniteurs de température et de tension, des moniteurs de vieillissement pourraient être intégrés de manière à corréler l'apparition des fautes intermittentes avec les mécanismes de vieillissement. En effet, il existe dans la littérature des moniteurs capables de déterminer le vieillissement induit par chaque mécanisme. Troisièmement, les différentes im-

plémentations de test périodique et de rétablissement des processeurs pourraient être intégrées ensemble, ce qui contribuerait à leur validation.

Prédiction du vieillissement Comme nous l'avons déjà mentionné, les fautes intermittentes sont dépendantes du vieillissement. Or, il est nécessaire d'affiner le lien entre l'activité créée par les applications, la température induite sur le processeur et son vieillissement. Nous avons montré que les processeurs soumis à un fort taux d'activité présentent un taux plus important d'erreurs intermittentes. Cette observation confirme le lien entre ces trois facteurs, cependant il doit être précisé. En particulier, le taux d'erreurs doit être observé à un grain plus fin, c'est à dire au niveau de la microarchitecture du processeur. Ainsi, s'il est possible d'affiner suffisamment cette étude, alors, il peut être envisageable de concevoir un modèle de vieillissement prenant en compte l'activité des processeurs, dans le but de prédire l'apparition des fautes.

Ces perspectives montrent que le problème des fautes intermittentes est vaste et que de nouvelles études sont nécessaires afin de cerner complétement le sujet. De plus, c'est un sujet d'actualité qui devrait s'intensifier dans les années à venir.

Glossaire

ASIC	*Application-Specific Integrated Circuit*
ATE	*Automatic Test Equipment*
BIST	*Built-In Self Test*
CFG	*Control-Flow Graph*, graphe de contrôle d'exécution
CMP	*Chip multiprocessor*
CUT	*Circuit Under Test*, circuit sous test
DFG	*Data-Flow Graph*, graphe de dépendance des données
DFT	*Design For Test*
DMR	*Dual-Modular Redundancy*
ECC	*Error Correction Code*, code correcteur d'erreurs
ELLF	*Enhanced Least-Laxity-First*
FIT	*Failure In Time*, nombre de défaillances par 10^9 h
GA	Générateurs d'Adresse
IO	Interface avec l'OSoC
ISA	*Instruction Set Architecture*
LFSR	*Linear Feedback Shift Register*, registre à décalage à rétroaction linéaire
MTBF	*Mean Time Between Failures*
MTTF	*Mean Time To Failure*
MTTR	*Mean Time To Repair*
NMR	*N-Modular Redundancy*
OSoC	*Operating System accelerator on Chip*
RMT	*Redundant Multitheading*
RTL	*Register Transfer Level*
SBST	*Software-Based Self-Test*
SCMP	*SCalable Multi-Processor*
SEC/DED	*Single-Error Correction/Double-Error Detection*
SESAM	*Simulation Environment for Scalable Asymmetric Multiprocessors*
SHS	*State History Signature*
SMT	*Simultaneous MultiThreading*
SoC	*System on Chip*
SoR	*Sphere of Replication*, sphère de redondance
TMAE	Temps moyen avant erreur
TMEDE	Temps moyen entre deux erreurs
TMEE	Temps moyen en erreur

TMR	*Triple-Modular Redundancy*, redondance triple
UGM	Unité de Gestion de la Mémoire
UI2O	Unité d'Interface et d'Interconnexion de l'OSoC
VLIW	*Very Long Instruction Word*

Bibliographie

[1] ITRS 2009 EDITION. *System Drivers*. Rap. tech. International Technology Roadmap for Semiconductors, 2009.

[2] J. ARLAT, Y. CROUZET, Y. DESWARTE, J.-C. FABRE et al. “Fault Tolerance”. Dans : *Encyclopedia of Computer Science and Information Systems*. 2006.

[3] I. KOREN et C. KRISHNA. *Fault-tolerant systems*. Elsevier/Morgan Kaufmann, 2007.

[4] J. SRINIVASAN, S. ADVE, P. BOSE et J. RIVERS. “The impact of technology scaling on lifetime reliability”. Dans : *DSN*. Florence, Italy, 2004, p. 177–186.

[5] S. MUKHERJEE, J. EMER et S. REINHARDT. “The soft error problem : an architectural perspective”. Dans : *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on* (2005), p. 243–247.

[6] P. M. WELLS, K. CHAKRABORTY et G. S. SOHI. “Adapting to intermittent faults in multicore systems”. Dans : *International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Seattle, WA, USA : ACM, 2008, p. 255–264.

[7] C. CONSTANTINESCU. “Intermittent faults and effects on reliability of integrated circuits”. Dans : *Reliability and Maintainability Symposium, 2008. RAMS 2008. Annual* (2008), p. 370 –374.

[8] H. QI, S. GANESAN et M. PECHT. “No-fault-found and intermittent failures in electronic products”. Dans : *Microelectronics Reliability* 48.5 (2008), p. 663 –674. URL : http://www.sciencedirect.com/science/article/B6V47-4SD1KTV-1/2/3f1cde215a2284cb9e50ab5e142b6a68.

[9] S. BORKAR, T. KARNIK, S. NARENDRA, J. TSCHANZ et al. “Parameter variations and impact on circuits and microarchitecture”. Dans : *Design Automation Conference (DAC)*. Anaheim, CA, USA : ACM, 2003, p. 338–342.

[10] C. CONSTANTINESCU. “Impact of Intermittent Faults on Nanocomputing Devices”. Dans : *Proc. IEEE/IFIP DSN (Supplemental Volume), Edinburgh, UK* (2007), p. 238–241.

[11] R. JOSEPH, D. BROOKS et M. MARTONOSI. “Control techniques to eliminate voltage emergencies in high performance processors”. Dans : *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*. 2003, p. 79 –90.

[12] M. POWELL et T. VIJAYKUMAR. “Exploiting resonant behavior to reduce inductive noise”. Dans : *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*. 2004, p. 288 –299.

[13] M. D. POWELL, M. GOMAA et T. N. VIJAYKUMAR. “Heat-and-run : leveraging SMT and CMP to manage power density through the operating system”. Dans : *SIGOPS Oper. Syst. Rev.* 38.5 (2004), p. 260–270.

[14] RENESAS TECHNOLOGY. *Semiconductor Reliability Handbook*. Rap. tech. Rev 1.01. Renesas Technology, Nov. 2008.

[15] S. RUSSELL, O. GONZALEZ, B. DE COUTERE et D. FURNISS. *High Availability Without Clustering*. Rap. tech. Number SG24-6216-00 in Redbooks. IBM Corp., 2001, 2001.

[16] A. MAHMOOD et E. MCCLUSKEY. “Concurrent error detection using watchdog processors-a survey”. Dans : *Transactions on Computers* 37.2 (1988), p. 160–174.

[17] J. SEGURA et C. F. HAWKINS. *CMOS Electronics : How it Works, how it Fails*. John Wiley & Sons, 2004.

[18] S. DUVALL. “Statistical circuit modeling and optimization”. Dans : *IEEE International Workshop on Statistical Metrology (IWSM)*. 2000, p. 56–63.

[19] K. BOWMAN, S. DUVALL et J. MEINDL. “Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration”. Dans : *IEEE Journal of Solid-State Circuits* 37.2 (2002), p. 183 –190.

[20] K. BOWMAN, A. ALAMELDEEN, S. SRINIVASAN et C. WILKERSON. “Impact of Die-to-Die and Within-Die Parameter Variations on the Clock Frequency and Throughput of Multi-Core Processors”. Dans : *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17.12 (2009), p. 1679 –1690.

[21] M. LUNDSTROM. *ECE 612 Lecture 13 : Threshold Voltage and MOSFET Capacitances*. 2006. URL : http://nanohub.org/resources/1855.

[22] M. GUPTA, J. RIVERS, P. BOSE, G.-Y. WEI et al. “Tribeca : Design for PVT variations with local recovery and fine-grained adaptation”. Dans : *MICRO 42 : Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2009, p. 435 – 446.

[23] S. MUKHERJEE. *Architecture design for soft errors*. Morgan Kaufmann, 2008, p. 1–38.

[24] JEDEC PUBLICATION. *Failure Mechanisms and Models for Semiconductor Devices*. Rap. tech. JEP122C. Jedec, Mar. 2003.

[25] T.-C. ONG, M. LEVI, P.-K. KO et C. HU. “Recovery of threshold voltage after hot-carrier stressing”. Dans : *Electron Devices, IEEE Transactions on* 35.7 (1988), p. 978 –984.

[26] Y. LI, Y. M. KIM, E. MINTARNO, D. GARDNER et al. “Overcoming Early-Life Failure and Aging for Robust Systems”. Dans : *Design & Test of Computers, IEEE* 26.6 (2009), p. 28–39.

[27] JEDEC PUBLICATION. *Foundry process qualification guidelines*. Rap. tech. JP001.01. Jedec, May 2004.

[28] L. RASHID, K. PATTABIRAMAN et S. GOPALAKRISHNAN. "Towards understanding the effects of intermittent hardware faults on programs". Dans : *International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2010, p. 101 –106.

[29] N. KRANITIS, A. MERENTITIS, N. LAOUTARIS, G. THEODOROU et al. "Optimal Periodic Testing of Intermittent Faults In Embedded Pipelined Processor Applications". Dans : *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings* 1 (2006), p. 1–6.

[30] JEDEC PUBLICATION. *Temperature, Bias, and Operating Life*. Rap. tech. JESD22-A108. Jedec, Jun. 2005.

[31] JEDEC PUBLICATION. *Highly Accelerated Temperature and Humidity Stress Test*. Rap. tech. JESD22-A110C. Jedec, Jan. 1999.

[32] JEDEC PUBLICATION. *Accelerated Moisture Resistance-Unbiased Autoclave*. Rap. tech. JESD22-A102C. Jedec, Jun. 2008.

[33] JEDEC PUBLICATION. *Temperature Cycling*. Rap. tech. JESD22-A104C. Jedec, May. 2005.

[34] JEDEC PUBLICATION. *High Temperature Storage Life*. Rap. tech. JESD22-A103C. Jedec, Nov. 2004.

[35] XILINX. *Device Reliability Report (UG116)*. Rap. tech. 2010. URL : http://www.xilinx.com/support/documentation/user_guides/ug116.pdf.

[36] AVNET. *AES-V5FXT-EVL30-G Evaluation Kit*. URL : http://www.em.avnet.com.

[37] IBM. *PowerPC 440*. URL : https://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_440_Embedded_Core.

[38] MINCO. *FLEXIBLE HEATERS DESIGN GUIDE*. URL : http://www.minco.com/.

[39] XILINX. *FPGA Virtex 5 FXT*. URL : http://www.xilinx.com/products/virtex5/fxt.htm.

[40] NATIONAL INSTRUMENTS. *LabView 7.1*. URL : http://ni.com/labview.

[41] M. GUTHAUS, J. RINGENBERG, D. ERNST, T. AUSTIN et al. "MiBench : A free, commercially representative embedded benchmark suite". Dans : *IEEE International Workshop on Workload Characterization*. 2001, p. 3–14.

[42] N. KRANITIS, A. PASCHALIS, D. GIZOPOULOS et G. XENOULIS. "Software-based self-testing of embedded processors". Dans : *Computers, IEEE Transactions on* 54.4 (2005), p. 461–475.

[43] L. WASSERMAN. *All of statistics : a concise course in statistical inference*. Springer Verlag, 2004, p. 312–319.

[44] D. SIEWIOREK et R. SWARZ. *Reliable Computer Systems : Design and Evaluation*. AK Peters, Ltd., 1998.

[45] F. AZAÏS, S. BERNARD, Y. BERTRAND, M. FLOTTES et al. *Test de Circuits et de Systèmes Intégrés*. Français. 11413. Collection EGEM, Ed.Hermès, 2004. URL : http://hal-lirmm.ccsd.cnrs.fr/lirmm-00109158/en/.

[46] R. WHITE et F. MILES. “Principles of fault tolerance”. Dans : *Applied Power Electronics Conference and Exposition, 1996. APEC ’96. Conference Proceedings 1996., Eleventh Annual*. T. 1. 1996, 18–25vol.1.

[47] S. REINHARDT et S. MUKHERJEE. “Transient fault detection via simultaneous multithreading”. Dans : *Computer Architecture, 2000. Proceedings of the 27th International Symposium on* (2000), p. 25–36.

[48] S. MUKHERJEE, M. KONTZ et S. REINHARDT. “Detailed design and evaluation of redundant multi-threading alternatives”. Dans : *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. 2002, p. 99–110.

[49] T. AUSTIN. “DIVA : a reliable substrate for deep submicron microarchitecture design”. Dans : *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*. 1999, p. 196–207.

[50] A. MEIXNER, M. BAUER et D. SORIN. “Argus : Low-Cost, Comprehensive Error Detection in Simple Cores”. Dans : *IEEE MICRO* 28.1 (2008), p. 52–59.

[51] S. SHYAM, K. CONSTANTINIDES, S. PHADKE, V. BERTACCO et al. “Ultra low-cost defect protection for microprocessor pipelines”. Dans : *ASPLOS-XII : Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM Press, 2006, p. 73–82.

[52] M. PSARAKIS, D. GIZOPOULOS, E. SANCHEZ et M. REORDA. “Microprocessor Software-Based Self-Testing”. Dans : *Design Test of Computers, IEEE* 27.3 (mai 2010), p. 4 –19.

[53] Y. LI, S. MAKAR et S. MITRA. “CASP : Concurrent Autonomous Chip Self-Test Using Stored Test Patterns”. Dans : *Design, Automation and Test in Europe, 2008. DATE’08* (2008), p. 885–890.

[54] D. ERNST, N. S. KIM, S. DAS, S. PANT et al. “Razor : a low-power pipeline based on circuit-level timing speculation”. Dans : *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. 2003, p. 7–18.

[55] E. MIZAN, T. AMIMEUR et M. F. JACOME. “Self-Imposed Temporal Redundancy : An Efficient Technique to Enhance the Reliability of Pipelined Functional Units”. Dans : *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on* (2007), p. 45–53.

[56] M. K. QURESHI, O. MUTLU et Y. N. PATT. *Microarchitecture-Based Introspection : A Technique for Transient-Fault Tolerance in Microprocessors*. 2004. URL : citeseer.ist.psu.edu/qureshi04microarchitecturebased.html.

[57] N. OH, P. SHIRVANI et E. MCCLUSKEY. “Error detection by duplicated instructions in super-scalar processors”. Dans : *Reliability, IEEE Transactions on* 51.1 (2002), p. 63–75.

[58] G. REIS, J. CHANG, N. VACHHARAJANI, R. RANGAN et al. “SWIFT : software implemented fault tolerance”. Dans : *Code Generation and Optimization, 2005. CGO 2005. International Symposium on* (2005), p. 243–254.

[59] N. J. WANG et S. J. PATEL. “ReStore : Symptom-Based Soft Error Detection in Microprocessors”. Dans : *IEEE Trans. Dependable Secur. Comput.* 3.3 (2006), p. 188–201.

[60] J. SOMERS. *Stratus ftServer–Intel Fault Tolerant Platform*. Rap. tech. Intel Developer Forum, Fall, 2002.

[61] A. WOOD, R. JARDINE et W. BARTLETT. "Data Integrity in HP NonStop Servers". Dans : *2nd IEEE Workshop on Silicon Errors in Logic and System Effects (SELSE), Urbana-Champaign*. 2006.

[62] T. SLEGEL, R. AVERILL III, M. CHECK, B. GIAMEI et al. "IBM's S/390 G5 Microprocessor Design". Dans : *IEEE MICRO* (1999), p. 12–23.

[63] F. SELLERS, M. XIAO et L. BEARNSON. *Error detecting logic for digital computers*. McGraw-Hill, 1968.

[64] G. REIS, J. CHANG, N. VACHHARAJANI, S. MUKHERJEE et al. "Design and evaluation of hybrid fault-detection systems". Dans : *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on* (2005), p. 148–159.

[65] H. INOUE, Y. LI et S. MITRA. "VAST : Virtualization-Assisted Concurrent Autonomous Self-Test". Dans : *Test Conference, 2008. ITC 2008. IEEE International*. 2008, p. 1–10.

[66] D. GIZOPOULOS, M. PSARAKIS, M. HATZIMIHAIL, M. MANIATAKOS et al. "Systematic Software-Based Self-Test for Pipelined Processors". Dans : *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 16.11 (2008), p. 1441–1453.

[67] M. BREUER. "Testing for intermittent faults in digital circuits". Dans : *Computers, IEEE Transactions on* 3 (1973), p. 241–246.

[68] S. KAMAL et C. PAGE. "Intermittent faults : a model and a detection procedure". Dans : *IEEE Transactions on Computers* (1974), p. 713–719.

[69] A. PASCHALIS et D. GIZOPOULOS. "Effective software-based self-test strategies for on-line periodic testing of embedded processors". Dans : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.1 (2005), p. 88–99.

[70] A. APOSTOLAKIS, D. GIZOPOULOS, M. PSARAKIS et A. PASCHALIS. "Software-Based Self-Testing of Symmetric Shared-Memory Multiprocessors". Dans : *Computers, IEEE Transactions on* 58.12 (déc. 2009), p. 1682 –1694.

[71] Y. LI, O. MUTLU et S. MITRA. "Operating system scheduling for efficient online self-test in robust systems". Dans : *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*. Nov. 2009, p. 201 –208.

[72] T. NAKAGAWA et K. YASUI. "Optimal testing-policies for intermittent faults". Dans : *Reliability, IEEE Transactions on* 38.5 (1989), p. 577–580.

[73] S. SU, I. KOREN et Y. MALAIYA. "A Continuous-Parameter Markov Model and Detection Procedures for Intermittent Faults". Dans : *Computers, IEEE Transactions on* C-27.6 (1978), p. 567–570.

[74] A. PASCHALIS et D. GIZOPOULOS. "Effective software-based self-test strategies for on-line periodic testing of embedded processors". Dans : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 1 (2004), 578–583 Vol.1.

[75] N. VENTROUX. "Contrôle en ligne des systèmes multiprocesseurs hétérogènes embarqués : élaboration et validation d'une architecture". Thèse de doct. Université de Rennes, CEA LIST, 2006.

[76] F. CHANG, C.-J. CHEN et C.-J. LU. “A linear-time component-labeling algorithm using contour tracing technique”. Dans : *Comput. Vis. Image Underst.* 93 (2 2004), p. 206–220. URL : http://portal.acm.org/citation.cfm?id=973388.973393.

[77] N. VENTROUX, A. GUERRE, T. SASSOLAS, L. MOUTAOUKIL et al. “SESAM : An MP-SoC Simulation Environment for Dynamic Application Processing”. Dans : *Computer and Information Technology, International Conference on* 0 (2010), p. 1880–1886.

[78] T. MURATA. “Petri nets : Properties, analysis and applications”. Dans : *Proceedings of the IEEE* 77.4 (2002), p. 541–580.

[79] J. HILDEBRANDT, F. GOLATOWSKI et D. TIMMERMANN. “Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems”. Dans : *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*. 1999, p. 208–215.

[80] O.S.C.INITIATIVE. “SystemC community”. Dans : *World Wide Web Document* (2006).

[81] R. AZEVEDO, S. RIGO, M. BARTHOLOMEU, G. ARAUJO et al. “The ArchC architecture description language and tools”. Dans : *International Journal of Parallel Programming* 33.5 (2005), p. 453–484.

[82] O. SINNEN. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007.

[83] J. SGALL. “On-line scheduling”. Dans : *Online Algorithms*. Sous la dir. d’A. FIAT et G. WOEGINGER. T. 1442. Lecture Notes in Computer Science. 10.1007/BFb0029570. Springer Berlin / Heidelberg, 1998, p. 196–231. URL : http://dx.doi.org/10.1007/BFb0029570.

[84] J. GUILHEMSANG, O. HERON, N. VENTROUX et A. GIULIERI. “Impact of the application activity on intermittent faults in embedded systems”. Dans : *IEEE VLSI Test Symposium (VTS)*. Dana Point, États Unis, 2011.

[85] O. HERON, J. GUILHEMSANG, N. VENTROUX et A. GIULIERI. “Analysis of On-Line Self-Testing Policies for Real-Time Embedded Multiprocessors in DSM Technologies”. Dans : *IEEE International On-Line Testing Symposium (IOLTS)*. Corfu Island, Greece, 2010.

[86] J. GUILHEMSANG, O. HERON, N. VENTROUX et A. GIULIERI. “On-line pseudo-periodic testing for embedded multiprocessor”. Dans : *European Test Symposium (ETS’10)*. Workshop-Paper, Session 6B : Fault Tolerance and Online Testing. Prague, République Tchèque, 2010.

[87] J. GUILHEMSANG, O. HERON, N. VENTROUX et A. GIULIERI. “Emphasis on the existence of intermittent faults in embedded systems”. Dans : *IEEE International Workshop on Defect & Data-Driven Testing (D3T)*. Austin, États Unis, 2010.

[88] J. GUILHEMSANG, O. HERON, N. VENTROUX et A. GIULIERI. “Aging Induced failures analysis in RISC-based processor cores”. Dans : *ACACES 2009 - Poster Abstracts*. Barcelona, Spain : High Performance, Embedded Architecture et Compilation (HIPEAC), 2009, p. 307–310.

Publications personnelles

Conférences internationales avec actes et comité de lecture

J. GUILHEMSANG, O. HERON, N. VENTROUX et A. GIULIERI. “Impact of the application activity on intermittent faults in embedded systems”. Dans : *IEEE VLSI Test Symposium (VTS)*. Dana Point, États Unis, 2011

O. HERON, J. GUILHEMSANG, N. VENTROUX et A. GIULIERI. “Analysis of On-Line Self-Testing Policies for Real-Time Embedded Multiprocessors in DSM Technologies”. Dans : *IEEE International On-Line Testing Symposium (IOLTS)*. Corfu Island, Greece, 2010

Conférences internationales sans actes et avec comité de lecture

J. GUILHEMSANG, O. HERON, N. VENTROUX et A. GIULIERI. “On-line pseudo-periodic testing for embedded multiprocessor”. Dans : *European Test Symposium (ETS’10)*. Workshop-Paper, Session 6B : Fault Tolerance and Online Testing. Prague, République Tchèque, 2010

Workshop avec comité de lecture

J. GUILHEMSANG, O. HERON, N. VENTROUX et A. GIULIERI. “Emphasis on the existence of intermittent faults in embedded systems”. Dans : *IEEE International Workshop on Defect & Data-Driven Testing (D3T)*. Austin, États Unis, 2010

Communications sans actes et sans comité de lecture

J. GUILHEMSANG, O. HERON, N. VENTROUX et A. GIULIERI. “Aging Induced failures analysis in RISC-based processor cores”. Dans : *ACACES 2009 - Poster Abstracts*. Barcelona, Spain : High Performance, Embedded Architecture et Compilation (HIPEAC), 2009, p. 307–310

Résumé

Aujourd'hui les systèmes embarqués sont partout et requièrent de plus en plus de puissance de calcul. Pour cela, ces dernières années, les progrès d'intégration ont permis de diminuer la taille des transistors, d'augmenter la fréquence de fonctionnement et de diminuer la tension d'alimentation. Tout cela en augmentant progressivement le nombre de processeurs dans une même puce. Mais cette évolution a un impact négatif sur la fiabilité.

D'une part, les systèmes deviennent de plus en plus sensibles à leur environnement, ce qui conduit à un taux plus important des fautes transitoires. D'autre part, les variations de procédés de fabrication dans les technologies actuelles, induisent de plus en plus de variations à l'intérieur même des composants. Ce phénomène, combiné à la hausse des tensions d'alimentation et de la température, augmente progressivement la probabilité d'occurrence des fautes intermittentes voire permanentes. De plus, le vieillissement des composants semble s'accélérer avec la diminution du facteur d'échelle et la diminution non uniforme des tensions d'alimentation, causant l'apparition prématurée des fautes intermittentes voire permanentes.

Si les fautes transitoires et permanentes sont étudiées depuis plusieurs années, ce n'est pas le cas des fautes intermittentes. Or, pour tenter de se prémunir de ces fautes, il est important de comprendre leur comportement, ainsi que leur impact sur le système et les applications. Pour cela, nous avons défini une plateforme expérimentale capable d'observer des erreurs intermittentes. Pour cela, des processeurs en technologie 65nm sont soumis, d'une part à une température supérieure à 160°C et d'autre part à l'exécution d'un ensemble d'applications. Nous avons ainsi, pu confirmer que les erreurs intermittentes peuvent être observées avant l'apparition d'erreurs permanentes. Nos résultats confirment la présence de défauts intermittents très tôt avant la période d'usure du circuit. De plus, nous montrons que les circuits intégrés soumis à la plus forte activité présentent le plus grand nombre d'erreurs intermittentes.

Cette étude confirme qu'il est possible d'observer des erreurs intermittentes et donc qu'il est possible de les détecter en ligne. Cependant, aucune solution de détection en ligne des erreurs proposées dans la littérature ne convient à la fois aux erreurs intermittentes et aux architectures multiprocesseur. Ainsi, nous avons développé une méthode de test périodique pour répondre à ces contraintes. En particulier, nous avons montré que le test ne doit pas nécessairement être prioritaire devant les applications. Cela nous a permis de conclure qu'une politique d'ordonnancement des tests pseudo-périodiques, prenant en compte les processeurs au repos et la priorité des tâches, offrent le meilleur compromis entre performance et probabilité de détection.

Mots-clés : tolérance aux fautes, fiabilité, fautes intermittentes, architectures multiprocesseur, détection en ligne des erreurs.